COMPUTER SCIENCE PUBLICATION

DECENTRALIZED REMAPPING OF DATA PARALLEL COMPUTATIONS WITH THE GENERALIZED DIMENSION EXCHANGE METHOD

Cheng-Zhong Xu and Francis C.M. Lau

Technical Report TR-94-01

January 1994

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF ENGINEERING
UNIVERSITY OF HONG KONG
POKFULAM ROAD
HONG KONG

Decentralized Remapping of Data Parallel Computations with the Generalized Dimension Exchange Method

Cheng-Zhong Xu and Francis C.M. Lau
Department of Computer Science, The University of Hong Kong

Abstract

The Generalized Dimension Exchange (GDE) method is a fully distributed load balancing method that is most suitable for multicomputers with a direct communication network. It is extremely easy to implement and can yield optimal performance given a proper tuning. We propose a decentralized remapping method that uses the GDE algorithm periodically to balance (remap) the system's load. We implemented this remapping method in two data parallel applications and found it to be effective in reducing the computation time. The gains in performance ($5-15\%$) due to remapping are reasonably substantial given the fact that the two applications by their very nature do not necessarily favor remapping.

1 Introduction

The mapping problem in parallel computations is to distribute the workload or processes of a computation across the available processors so that each processor would end up with more or less the same amount of work to do. In most cases, this is done prior to execution and is done only once—called *static* mapping. Static mapping can be quite effective for computations that have predictable runtime behaviors [8]. For computations whose runtime behavior is non-deterministic or not so predictable, however, doing mapping only once in the beginning is insufficient. For these cases, one might have to do the mapping more than once or periodically during runtime—this is called *dynamic remapping*. Dynamic remapping produces ideal load-balances at the cost of additional runtime overheads. A successful remapping mechanism must therefore try to produce enough benefits that would outweigh the overheads incurred. We introduce such a remapping mechanism in this paper, which is based on a very simple but effective load balancing method, the *Generalized Dimension Exchange* (GDE) method [27, 26]. We demonstrate the effectiveness of this mechanism through its application to two major applications.

A data parallel computation decomposes its problem domain into a number of subdo-

mains (data sets), and designates them to processes [8]. These processes simultaneously perform the same functions across different data sets. Because the subdomains are connected at their boundaries, processes in neighboring subdomains have to synchronize and exchange boundary information with each other every now and then. These synchronization points divide the computation into phases. During each phase, every process executes some operations that might depend on the results from previous phases. This kind of computations arises in a large variety of real applications. In a study of 84 successful parallel applications in various areas, it was found that nearly 83% used this form of data parallelism [7].

In data parallel computations, the computational requirements associated with different parts of a problem domain may change as the computation proceeds. This occurs when the behavior of the physical system being modeled changes with time. Examples include parallel iterative solutions of time-dependent partial differential equations [1], time-driven discrete event simulations [18], and parallel image processing [4]. To implement this kind of computations in a multicomputer, *static* domain decomposition techniques, such as stripwise, boxwise, and binary decompositions [2], often fail to maintain an even spread of computational workloads across the processors during execution. Because of the need of synchronization between phases, a processor that has finished its work in the current phase has to wait for the more heavily loaded processors to finish their work before proceeding to the next phase (see Figure 2). Consequently, the duration of a phase is determined by the heavily loaded processors, and system performance may deteriorate in time.

To lessen the penalty due to synchronization and load imbalances, one must dynamically remap (re-decompose) the problem domain onto the processors as the computation proceeds. Remapping can be performed either from scratch—*i.e.*, treating the current overall workload as if it is a brand new workload to be decomposed—or through adjusting boundaries created in the previous decomposition. The former approach can be viewed as dynamic invocation of a static decomposition. Since the global workload is to be taken as a whole for re-decomposition, the work is most conveniently performed by a designated processor which has a global view of the current state of affair. Such a centralized remapping can no doubt yield a good workload distribution because of the existence of global knowledge. However, the price to pay is the high cost of collecting the data sets from and communicating the re-decomposed data sets to the processors, which could be prohibitive, especially in large scale systems. Therefore, the second approach of adjusting boundaries from the previous phase, which can easily be performed in a decentralized, parallel fashion, is preferred. As each processor has to deal only with its nearest neighbors, much fewer data transfers would take place in the network as compared to the centralized approach. The difficulty lies in how to decide in a distributed way when a remapping should be invoked and how to do

the adjusting of the subdomain boundaries among processors (also in a distributed way) so that the result is a reasonably balanced workload.

In the literature, much attention has been given to dynamic remapping of data parallel computations in recent years. Nicol *et al.* addressed the issue of deciding when to invoke a remapping so that its performance gain will not be offset by its overhead [19, 18]. They proposed two simple but effective heuristical invocation policies for two kinds of workload models: one in which the computational requirements of a region evolve gradually, and one abruptly. With these policies, the invocation decision is made in a centralized manner and based on an assumption that the remapping cost is known in advance. Albeit valid in centralized remapping, the assumption is obviously not applicable to decentralized remapping.

De Keyser and Roose experimented with centralized remapping in the calculation of dynamic unstructured finite element grids [14, 15]. The computation is solution-adaptive in that the grids are refined according to the solution obtained so far. After the refinement of the grids, a global remapping is imposed. Dynamic remapping for solution-adaptive grid refinements was also considered by Williams [25]. He compared three complex parallel algorithms for doing the remapping, the orthogonal recursive bisection, the simulated annealing, and the eigenvalue recursive bisection, and concluded that the last one should be preferred.

Choudhary *et al.* incorporated a remapping mechanism into a parallel motion estimation system [4, 3]. The system consists of several stages: convolution, thresholding and template matching. Remapping is invoked at the beginning of each stage, in which every processor would broadcast information about the subdomain it is working on to all others, and then do border adjustment based on the collected information. A similar idea based on global knowledge was implemented by Hanxleden and Scott [10]. They invoked remapping periodically in the course of a Monte Carlo dynamical simulation, and gained $10 - 15\%$ performance improvement using the optimal remapping interval.

This paper proposes a new and effective dynamic remapping method for time-dependent data parallel computations. This method is fully distributed and is based on a very simple, low-overhead load balancing algorithm that runs in every node. We prefer a distributed method because it would have less chances of running into bottleneck problems and is generally more reliable. The method requires no global information, and there is no broadcasting of information. Through some actual implementations (WaTor simulation and parallel thinning of images), the net gain in performance due to the use of this method is found to be reasonably substantial and comparable to performance results in the literature for decentralized remapping. The simple load balancing algorithm that runs in every node is based on the Generalized Dimension Exchange (GDE) load balancing method. With the GDE method, each processor plays an identical role in making load balancing decisions, which is

based on knowledge of its nearest neighbors' states. GDE load balancing is iterative: every processor successively balances its workload with each one of nearest neighbors in an iteration step until a global balanced state is reached. This global balanced state is detected by a distributed termination detection algorithm embedded in the load balancing algorithm. We have analyzed the GDE method rather thoroughly in our previous works, and showed that that it is highly effective, scalable, and applicable to many network topologies [27, 26]. This paper on the one hand continues the study of the GDE method with emphasis on its applicability to real problems, and on the other hand tries to demonstrate the benefits of distributed remapping in general.

The rest of paper is organized as follows. Section 2 describes the computation model and reviews the GDE method. Section 3 presents implementation aspects of our GDE-based remapping mechanism. Section 4 evaluates the performance of the mechanism in two different data parallel applications. We make a conclusion in Section 5.

2 The computation model and the GDE method

2.1 The computation model

We consider time-dependent data parallel computations in multicomputers. A multicomputer is assumed to consist of N autonomous, homogeneous processors connected by a direct communication network. We represent the network by a simple graph $G = (V, E)$, where V denotes the set of processors labeled from 1 to N, and $E \subseteq V \times V$ is a set of edges. Each edge $(i, j) \in E$ corresponds to the communication link between processors i and j. Figure 1 presents several examples of popular network topologies.

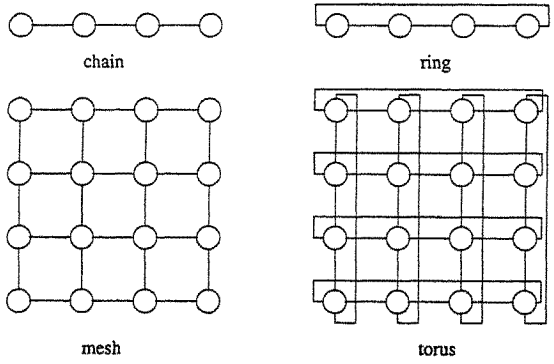

Figure 1: Examples of network topologies

The parallel computation is assumed to follow the so-called Single-Program-Multiple-Data (SPMD) paradigm in which each processor executes the same program but on different subdomains of the problem [8]. It proceeds in phases that are separated by global synchronization points. During each phase, the processors perform calculations independently and then communicate with their data-dependent peers. Figure 2 shows a typical scenario of the paradigm in a system of four processors. The horizontal scale corresponds to elapsed time (or computation time) of the processors; the vertical lines represent synchronization points at which a round of communications among the processors is due to begin. The shaded and the dark horizontal bars repressent the communication time and the calculation time respectively. The dotted empty bars correspond to the idle times of the processors, which are the times spent in waiting for the next phase to come.

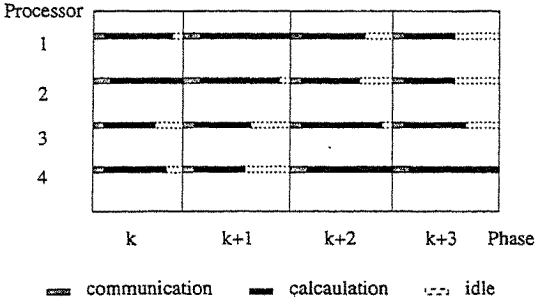

Figure 2: A example of time-dependent multiphase data parallel computations

Let t_{ik}^{comm}, t_{ik}^{calc} and t_{ik} denote the communication time, the calculation time, and the elapsed (or computation) time of processor i in the k^{th} phase, respectively. Let T_k denote the duration of the k^{th} phase. Then, the computation time of the processor in this phase is given by

$$t_{ik} = t_{ik}^{comm} + t_{ik}^{calc} \tag{1}$$

and the duration of the phase by

$$T_k = \max(t_{1k}, t_{2k}, \ldots, t_{Nk}). \tag{2}$$

Supposing that the computation requires K phases, then the total elapsed time of the computation, denoted by T, is given by

$$T = \sum_{k=1}^{K} T_k. \tag{3}$$

Clearly, the efficiency of this kind of computations is dependent upon two main factors: the interprocessor communication cost and the degree of load imbalances across the

processors during each phase. The interprocessor communication cost is reflected by the ratio of t_{ik}^{comm} to t_{ik}^{calc}. These two times are functions of various parameters related to the application in question as well as the underlying multicomputer system. Discussion of the problem of how to tune these parameters to yield a better communication-calculation ratio is beyond the scope of this paper. Readers are referred to the book by Cox *et al.* [8] for discussion on this issue and practical techniques to use. In this paper, we assume that the computation is dominated by their calculation times which is a reasonable assumption for medium- and large-grain data parallel computations. As such, we can base the calculation of load imbalances on calculation times. These calculation times are in fact equal to processors' utilization, which is defined as

$$U_k = \frac{\sum_{i=1}^{N} t_{ik}}{NT_k}. \tag{4}$$

The objective of remapping is to minimize the total elapsed time through maximizing the processor utilization U_k from phase to phase.

Since the execution of the remapping procedure is expected to incur non-negligible delay, the decision of when to invoke a remapping must be carefully made so that the remapping cost would not outweigh the performance gain. Notice that the distribution of the computation times t_{ik} have much bearing on the reward of remapping. Given some time-dependent computation whose execution behavior is non-deterministic, it is possible for the distribution of t_{ik}, $1 \leq i \leq N$, in a phase to tend to uniform; and a uniformly distributed t_{ik} in some phase to become severely imbalanced in the next phase. That is, for computations whose execution behavior is non-deterministic (and unpredictable), the possibility that a version with no balancing would outperform a dynamically balanced version exists, regardless of how well one can optimize the remapping procedure. Therefore, in order for remapping to be promising in leading to appreciable performance gains, the ideal arena in which to apply remapping would be the class of parallel computations whose computational requirements vary gradually over time. There are a large number of practical examples that fall into this class, including the two we implemented.

2.2 The GDE method

Our remapping mechanism invokes the GDE load balancing procedure between every two successive phases, k and $k+1$ say, so that hopefully an averaged computation time $t_{i(k+1)}$ may result across the processors. For simplicity, we use t_i to represent the expected computation time $t_{i(k+1)}$ of processor i prior to remapping. The goal of remapping with the GDE method is to redistribute the system workload such that each processor would end up with the same expected computation time $t = \sum t_i/N$. Such a redistribution is possible if

the computation time t_i is dominated by its calculation part t_i^{calc} and if t_i^{calc} is equivalent to the fact that there are t_i^{calc} pieces of work, each requiring one unit of execution time. If that's the case, we can count the outstanding pieces of work in a processor at the end of a phase and use this number to calculate the average load to be assigned to every processor in the next phase.

The GDE method evolves from the Dimension Exchange (DE) method which was intensively studied in hypercube-structured multicomputers [5, 13, 20, 21]. With the DE method, every processor successively balances its workload with each one of its direct neighbors in an iteration step according to a dimension order of the hypercube. It was proved that regardless of the dimension order, the DE method yields a uniform distribution from any initial workload distribution after one round of iteration steps (called a sweep) [5]. For non-hypercube structures, however, we proved that the DE method is not most efficient (not optimal). Subsequently we derived the generalized version, the GDE method, which can yield optimal results in non-hypercube networks. The GDE method is based on edge-coloring of the interconnection graph $G = (V, E)$. The edges of G are supposed to be colored beforehand with the least number of colors (κ, say), and no two adjoining edges are assigned the same color number. We index the colors with integers from 1 to κ. Figure 3(a) shows a colored 4×4 mesh. The numbers in parentheses are the assigned color numbers (or chromatic indices).

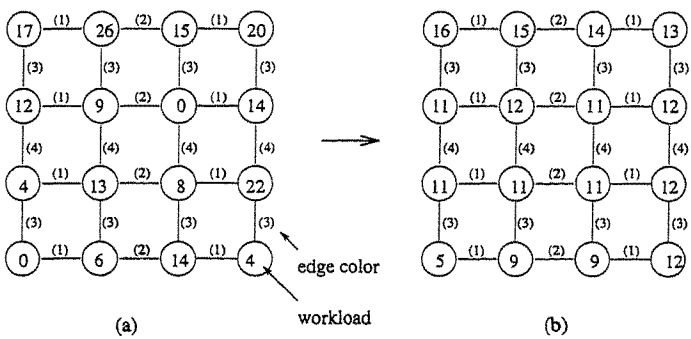

Figure 3: Change of workload distribution after an iteration sweep

In an edge-colored graph, a "dimension" is then equal to the set of all edges of the same chromatic index. During each iteration sweep, a processor would exchange its load with each one of its neighbors in turn according to the chromatic indices of its incident edges—*i.e.*, going through the dimensions in turn. During the process, the exchange operation between a pair of processors would split the total workload of the two processors according to a prescribed *exchange parameter* λ. Specifically, for processor i, the exchange of workload

with a nearest neighbor j is executed as

$$\tilde{t}_i = (1-\lambda)t_i + \lambda t_j \tag{5}$$

where t_i and t_j are the current workloads of processor i and j respectively, and $\tilde{t}_i$ is the adjusted workload of processor i; processor j executes a similar operation. Note that when $\lambda = 1/2$, the GDE method is reduced to the original DE method [5, 13]. Figure 3 illustrates the change of workload distribution subject to the integer ceiling and floor operations after an iteration sweep of the DE method.

Clearly, for an arbitrary structure, it is unlikely that the DE method can yield a uniform workload distribution after a single iteration sweep. The introduction of the exchange parameter λ aims at accelerating the convergence rate of the load balancing procedure. Our previous work resulted in the determination of a necessary and sufficient condition for values of this parameter that would lead to convergence. We also presented a sufficient condition for a class of network topologies for which $\lambda = 1/2$ yields the fastest convergence rate. Examples of the members of this class include the hypercube and the product of any two structures satisfying the condition.

For the popular structures of the n-dimensional $k_1 \times k_2 \times \cdots \times k_n$ mesh and $2k_1 \times 2k_2 \times \cdots \times 2k_n$ torus, we derived the optimal exchange parameters, $\lambda_{opt} = 1/(1+\sin(\pi/k))$, where $k = \max\{k_i, 1 \leq i \leq n\}$. Note that the torus converges twice as fast as a mesh of the same dimensions. Through extensive experimentation, we showed that these optimal parameters did speed up the GDE balancing procedure significantly, and that the actual number of iteration steps taken was encouragingly small. And we find that the performance of these optimally tuned procedures is quite scalable: the number of iteration steps is found to be $O(N)$, where N is the number of processors.

3 Distributed remapping with the GDE method

3.1 Distributed convergence detection

The distributed GDE load balancing procedure is invoked between consecutive phases. Everytime when it finishes, the variance of the global workload distribution across the processors is supposed to be less than a certain prescribed threshold. Such a state has to be detected in order for the procedure to stop executing. From the practical point of view, the detection of the global convergence is by no means trivial because the processors are unaware of the global workload distribution during balancing. To assist the processors to infer global termination of the load balancing procedure from local workload information, we superimpose a distributed termination detection mechanism on the load balancing

procedure.

The problem of distributed termination detection per se is a popular research topic in parallel and distributed computing realms. In the past, numerous solutions of diverse characteristics have been proposed [9, 16, 22, 28]. For our situation, we need a fully distributed and highly efficient method for the nodes to detect global termination because all the processors are held up waiting for the completion of the remapping procedure before they can proceed into the next phase. The method's delay in announcing termination after the instant when every node enters into its locally terminated (or stable) state must be sufficiently small. A stable state is one in which the workload remains unchanged after an iteration sweep of the GDE load balancing. We adopt the method as proposed in [28], which is especially effective for termination detection of loosely synchronized iteration computations in general.

The method (see the algorithm below) makes use of global virtual time advanced by the iteration sweeps. For termination detection, every node maintains an integer counter *State* to record its current state and the historical states of others. *State* is equal to zero if and only if the node is in an unstable state. Every node exchanges its counter value with its nearest neighbors in a manner that is exactly like the exchange of workload information in the GDE method. By executing the operation *Exchange(c)*, a node sends out its local counter value and receives its neighbor's counter value along the channel with chromatic index c. The variable *InputState* temporarily stores the neighbor's counter value received in the current exchange operation. The counter *State* changes as the load balancing procedure, *LoadBalance*(), progresses. Global termination of the load balancing procedure is detected when the counter value *State* reaches a prescribed value, Δ, which is a function of the structure of the underlying colored system graph. In [28], we showed that this Δ is in fact equal to the minimum number of *sweeps* required by a processor to obtain knowledge of others' statuses. In the color mesh of Figure 3(a), for example, $\Delta = 2$, because it takes a minimum of two sweeps for any node to transmit a message to any other node. As a counter would keep counting up after the system has entered a global stable state, this method is time-optimal.

Algorithm: *TerminationDetector*

$State = 0$;
while $(State \leq \Delta)$ {
  for $(c = 1; c \leq \kappa; c++)$ {
    if there exists an incident edge of color c {
      $InputState = Exchange(c, State)$;
      $State = \min\{State, InputState\}$;

```
        }
    }
    LoadBalance();
    if (LocalTerminated)
        State = State + 1;
    else
        State = 0;
}
```

3.2 Implementation of a GDE-based remapping mechanism

Even though the analysis of the GDE method ignores the interprocessor communication overhead, the method is applicable to the computations in practice whose interprocessor communication cost is non-negligible. Basically, what the GDE load balancing ends up with are near-neighbor communications that shift the loads across short distances. By adopting the approach of adjusting boundaries of the problem domain, the GDE balancing procedure preserves the communication locality and hence the stability of the original communication structure through the series of re-decompositions.

The problem domain can be treated as a group of *internal load distributions* together with a corresponding *external load distribution*. Every subdomain in a node is represented by an internal load distribution for the computational requirements of its internal finer portions, and by an external integer value for its total workload. Table 1 shows an example of the internal and external load distributions of a 64×64 problem domain. The domain is supposed to be distributed in strips across eight processors connected as a chain. Each strip is made up of 8 rows (*i.e.*, internal finer portions of a strip). We take a row as the basic unit in the re-decomposition, and attach to it an external integer value to represent its computational requirement.

The remapping mechanism has two components: the *decision maker* and the *workload adjuster*. The decision maker is concerned only with the external load distribution, and is responsible for calculating the amount of workload inflow or outflow along each link of a node necessary for workload balancing. The workload adjuster is responsible for actually adjusting the borders of the problem domain according to the results of the decision maker.

The decision maker uses the GDE method with which a node iteratively balances its workload with its nearest neighbors until a uniform workload distribution is reached and detected. Note that the balance operator does not involve real workload. The workload is represented abstractly in this decision making process by simple integer variables: *WorkLoad* for the node's workload before the decision making and *Load* a temporary variable for

Table 1: Load distributions of a problem domain

Node index	Internal distribution	External load
1	(11, 15, 16, 18, 19, 20, 22, 21)	140
2	(21, 22, 21, 22, 21, 21, 22, 23)	173
3	(24, 22, 23, 23, 23, 24, 24, 22)	189
4	(20, 19, 18, 18, 18, 19, 20, 21)	154
5	(28, 28, 26, 28, 32, 33, 34, 37)	248
6	(34, 24, 22, 21, 21, 17, 17, 17)	171
7	(16, 14, 14, 16, 17, 16, 17, 18)	127
8	(16, 15, 14, 14, 13, 11, 11, 11)	106

workload during the process. We also introduce a vector $FlowTrace$ to keep track of the workload flows along each link of a node. Initially, each element $FlowTrace[i]$ is set to zero. For each sweep of the iterative decision making procedure, the amount of workload which is to be sent away or absorbed along a link i is added to $FlowTrace[i]$ as a positive or negative value respectively. Thus, at the end of the decision making, $FlowTrace$ records the inflow or outflow amount along each link.

Below, we outline the algorithm of the decision maker, which combines the GDE algorithm and the termination detection algorithm. $LocalTerminated$ becomes true in processor i when no change occurs in $FlowTrace[i]$ after a sweep of exchanges with the processor's neighbors.

Algorithm: *DecisionMaker*

$State = 0$;
$Load = WorkLoad$;
while $(State \leq \Delta)$ {
  for $(c = 1; c \leq \kappa; c++)$ {
    if there exists an edge of color c {
      $(InputState, InputLoad) = Exchange(c, State, Load)$;
      if $(InputLoad > Load)$
        $temp = \lfloor (InputLoad - Load) \times \lambda \rfloor$;
      else
        $Flow[c] = \lceil (InputLoad - Load) \times \lambda \rceil$;
      $FlowTrace[c] = FlowTrace[c] + temp$;
      $Load = Load + temp$;

```
                State = min{State, InputState};
            }
        }
        if (LocalTerminated)
            State = State + 1;
        else
            State = 0;
    }
```

Applying the algorithm to the external load distribution in Table 1, we obtain the inflow or outflow value for each link, as illustrated in Figure 4.

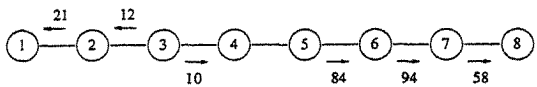

Figure 4: Illustration of in/out-flow along each link

Following the decision making, the workload adjuster of every node would start to work on its internal load distribution according to the *FlowTrace* vector generated by the decision maker. It involves selecting data set(s) to be split, transferring the split data sets between nodes, and merging of received data set(s) with the original subdomain. In principle, the splitting and merging of data sets are such that the geometric adjacency of data points in the problem domain are kept intact—basically adjusting of borders. For stripwise-partitioned subdomains such as those in Table 1, this can be done rather conveniently through shifting rows (or columns) between neighboring subdomains. Details of how this is done will be presented in the next section when the implementations of two real applications are discussed.

4 Performance evaluation

The time-dependent data parallel applications we have chosen for testing the performance of the GDE-based remapping mechanism are the WaTor simulation [6] and parallel thinning of images [12, 11]. They are representatives of two different load variation models: in the Wator simulation, computation requirements in a phase are independent of the previous phase, whereas in image thinning, the computation requirements of a phase are dependent on the previous phase. The experiments were coded in INMOS Parallel C [24], and run on a group of T805-30 transputers [23]. The main metric of interest is the *improvement* in

execution time due to remapping, denoted by η_{remap}, which is defined as

$$\eta_{remap} = \frac{t_{WithoutRemap} - t_{WithRemap}}{t_{WithoutRemap}} \times 100\%$$

where $t_{WithoutRemap}$ and $t_{WithRemap}$ are the execution times without and with remapping respectively. Another metric of interest is the overhead of the remapping mechanism. The timings are expressed in ticks. A tick is the basic timing unit in transputer and is equivalent of 64 microseconds.

4.1 WaTor—A Monte Carlo dynamical simulation

WaTor is an experiment that simulates the activities of fishes in a two-dimensional periodic ocean. The name WaTor comes from the toroidal topology of the imaginary watery planet. Fishes in the ocean breed, move, eat and die according to certain non-deterministic rules. The simulation is Monte Carlo in nature, and can be used to illustrate many of the crucial ideas in dynamic time- and event-driven simulations.

In the simulation, the ocean space is divided into a fine grid which is structured as a torus. Fishes are allowed to live only on grid points, and move around within neighboring points in a simulation step. There are two kinds of fishes in the ocean: the *minnows* and the *sharks*. They adhere to the following rules as they strive to live.

1. Each fish is updated as the the simulation progresses in a series of discrete time steps.
2. A minnow locates a vacant position randomly in up, down, left or right direction. If the vacant position is found, the minnow moves there, and leaves a new minnow of age 0 in the original location if it is mature with respect to the *minnow breeding age*; otherwise, the minnow stays where it is and cannot breed regardless of its age.
3. A shark locates a minnow within its neighboring positions at first. If found, it eats the minnow and moves to that location; otherwise, the shark then locates a vacant position like what the minnow does. If the shark moves to a new location and it is mature with respect to the *shark breeding age*, a new shark of age 0 is left in its original location. If a shark has not eaten any minnows within a *starvation* period, it dies.

Since the ocean structure is toroidal, we implement the WaTor algorithm on a ring-structured system (a ring is a special case of a torus). The parallel implementation decomposes the ocean grid into strips, and assigns each of them to a processing node. Each simulation step consists of three substeps:

1. *ExBound*: exchange of contents of grid points along a strip's boundaries;

2. *Update*: update of fishes in the strip;

3. *ExFish*: boundary-crossing of fishes that have to leave the strip.

The routine *Update* follows the above live-and-die rules for fishes. Fishes bound for neighboring strips are not transferred individually. Instead, they are held until the end of the *Update* procedure, and then bundled up to cross the boundaries in the routine *ExFish*. Thus, the duration of the a simulation step in a processor i is given by

$$T_i = t_i^{ExBound} + t_i^{Update} + t_i^{ExFish}$$

where $t_i^{ExBound}$, t_i^{Update} and t_i^{ExFish} are the times spent in the respective procedures. Since the boundary-crossing fishes are bundled and transferred in one message, every processor transmits the same number of messages to its neighbors and hence incurs approximately the same communication cost.

The simulation is done for a 256×256 ocean grid which is mapped onto 16 transputers. And it is run for as many as 100 simulation steps. The *minnow breeding*, the *shark breeding* and the *shark starvation* parameters are set to be 7, 12, and 5 steps respectively, Initially, the ocean is populated by minnows and sharks generated from a random uniform distribution, which are distributed by rows among the 16 transputers. The total simulation time for 100 steps is 274716 ticks.

The computational requirement of a processor is proportional to the density of fishes in the strip that it is simulating. More fishes exist, more computation time is needed for the update. Owing to the tendency of the fishes to form schools dynamically and unpredictable breeding, eating and dying of the fishes, the processor utilization U_k (as defined in Section 2) of the processors varies not so smoothly over time, as shown in the TIME curve of Figure 5. We apply remapping based to the GDE method periodically on the parallel simulation. Since the computational workload of a processor is proportional to the number of fishes in the strip, we use the latter as the measure of workload. The remapping procedure then tries to split the number of fishes between nodes as evenly as possible. Figure 5 (the FISH curve) also plots the processor utilization in terms of the number of fishes of each processor at various simulation steps. From the simulation data, it is observed that the variance of the computation time distribution changes with time and the change tendency is unpredictable. The close agreement in shape of the two curves confirms the reasonableness of measuring the computational load in terms of the number of fishes.

We examine the benefits of GDE-based remapping for various sizes of the remapping interval. Other than the remapping interval, the remapping cost relative to the computation

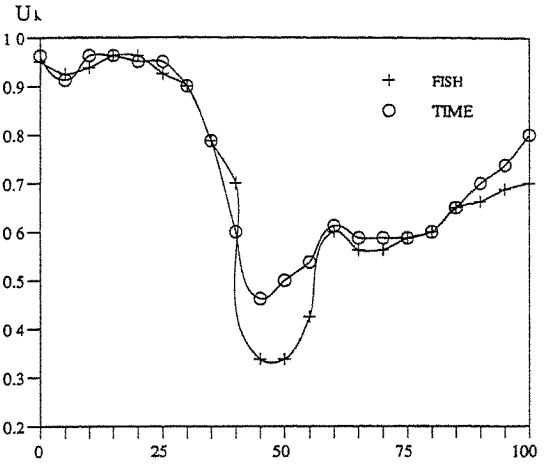

Figure 5: Processor utilization at various simulation steps

time per simulation step is also an important parameter for determining the improvement due to remapping. The remapping cost is the time $t_i^{decision}$ required for decision-making, plus the time t_i^{adjust} spent in the workload adjustment. The latter is dependent on the internal fish distribution of processors at the time the remapping is invoked and the results of the decision making. Table 2 presents the scenario of a remapping instance which is invoked after the simulation has passed 30 steps. The distribution of number of fishes f_i^{pre} prior to the remapping is given in the second column. For comparison, we also record in the next column the actual update time of each processor, t_i^{update}, if no remapping is imposed. The fourth column presents the time $t_i^{decision}$, which is the product of the number of iteration sweeps NS spent in decision making using the GDE method (among which are 8 sweeps for global convergence detection) and a constant representing the time complexity of a sweep. The fifth column gives the number of fishes that are migrated upwards (f_i^{up}) and downwards (f_i^{down}) due to the remapping. A positive number means "take" and a negative number means "give away". Correspondingly, the time spent in load adjustment t_i^{adjust} is presented in the sixth column. The last column is the distribution of fishes after remapping, f_i^{post}.

It can be seen from the table that the time for decision making $t_i^{decision}$ (= 43) is relatively insignificant when compared with the update time t_i^{update} (1900 on average), and the cost of remapping is dominated by the time for load adjustment t_i^{adjust}. It is because the load adjustment procedure involves a number of time-consuming steps including memory allocation/deallocation and transmission of the fishes concerned.

Table 2: Scenario of a remapping instance

	pre-remapping		decision-making	load adjustment		post-remapping
p_i	f_i^{pre}	t_i^{update}	$(NS, t_i^{decision})$	(f_i^{up}, f_i^{down})	t_i^{adjust}	f_i^{post}
1	2014	4155		(-121, 0)	583	1893
2	2094	4464		(0,0)	2	2094
2	1909	3762		(0,0)	2	1909
4	1927	3974		(0,0)	2	1927
5	2160	4322		(0, -126)	528	2034
6	2123	4277		(126, -228)	1120	2021
7	1737	3518		(228, 0)	1116	1965
8	1727	3651	(17, 43)	(0, 109)	393	1836
9	2042	4235		(-109, 0)	393	1933
10	2132	4194		(0, 0)	2	2132
11	2017	3997		(0, -117)	545	1900
12	1876	4069		(117, 0)	543	1993
13	1743	3612		(0, 0)	2	1743
14	1848	3923		(0, 122)	479	1970
15	1931	3897		(-122, 139)	647	1948
16	2030	4165		(-139, 121)	646	2012

The improvements due to remapping for different sizes of the remapping interval are plotted in Figure 6 (the curve of DR,EL=0; where DR stands for distributed remapping, and EL stands for extra workload). The horizontal scale is the number of simulation steps between two successive remapping instances; for example, a 20 means remapping is invoked once every 20 simulation steps. The curve shows that relatively frequent remapping gives an improvement of $7 - 15\%$ in the overall simulation time. Because of its rather low cost, remapping is favorable even in the case that the simulation is interrupted by remapping once every simulation step. Conversely, less frequent remapping (*e.g.*, once every 30 to 50 steps) could end up with no improvement or even degradation of performance. It is because the load distribution across processors changes in a haphazard way over time. This is characteristic of WaTor simulations.

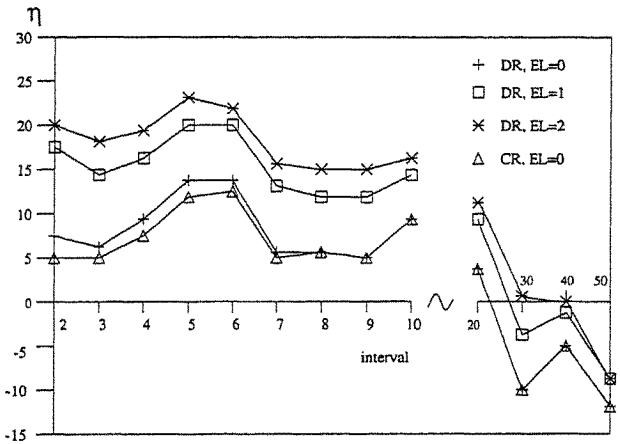

Figure 6: Improvement due to remapping for various interval sizes

Since the decision making and the load adjustment are based on the number of fishes, the remapping cost does not change with the increase of the time a fish spends in a simulation step. Hence, remapping would be even more beneficial if a fish does some extra work in a simulation step such as spending more time in its moving, eating, breeding and dying. This "better" performance is shown in Figure 6: the cases of DR,EL=1 and DR,EL=2, in which the time for the update of a fish is increased by an extra load of 1 and 2 ticks respectively.

For comparison, we also give improvements resulting from an efficient implementation of a centralized remapping method in the figure (the case of CR,EL=0; CR stands for centralized remapping). With the method, a designated processor takes the responsibility of making decisions according to the external load distribution [10]. This centralized version

takes a much longer time (100-200 ticks) to do decision making than the decentralized version based on GDE. It can be measured that the GDE-based remapping method, when frequently invoked, outperforms the centralized method by a factor of 40% or so.

4.2 Parallel thinning of images

Thinning is a fundamental preprocessing operation to be applied over a binary image to produce a version that shows the significant features of the image (see Figure 7). In the process, information that is deemed redundant is removed from the image. It takes as input a binary picture consisting of objects and a background which are represented by 1-valued pixels and 0-valued pixels respectively. It produces object skeletons that preserve the original shapes and connectivity. An iterative thinning algorithm performs successive iterations on the picture by converting those 1-valued pixels that are judged to be not belonging to the skeletons into 0-valued pixels until no more conversions are necessary. In general, the conversion (or survival) condition of a pixel, say P, is dependent upon the values of its eight neighboring pixels, as depicted below.

$$\begin{array}{ccc} NW & N & NE \\ W & P & E \\ SW & S & SE \end{array}$$

A parallel thinning algorithm decomposes the image domain into a number of portions and applies the thinning operator to all portions simultaneously. Since the study of parallel thinning algorithms itself is beyond the scope of this study, we arbitrarily picked an existing parallel thinning algorithm, the HSCP algorithm [12], and implemented it on a chain-structured system using stripwise decomposition. The algorithm is sketched below.

Algorithm: Thinning

```
while (!GlobalTerminated) {
    ExBound();
    if (!LocalTerminated) {
        ComputeEdge();
        ExEdge();
        LocalTerminated = Erosion();
    }
}
```

At the beginning of each thinning iteration step, the boundary pixels of each strip are exchanged with those of the neighboring strips in the routine *ExBound*. The heart of the

algorithm is the routine *Erosion* which applies the thinning operator to each pixel according to the following survival condition.

$$
\begin{aligned}
p \;\&\&\; & (!edge(P) \;||\\
& (edge(E) \;\&\&\; n \;\&\&\; s) \;||\\
& (edge(S) \;\&\&\; w \;\&\&\; e) \;||\\
& (edge(E) \;\&\&\; edge(SE) \;\&\&\; edge(S)))
\end{aligned}
$$

where a small letter denotes the pixel value at a location identified by the corresponding capital letter; the function *edge* tells whether a pixel is on the edge of an object skeleton. The *edge value* of a pixel is determined by the values of its surrounding pixels, and is computable in advance. The routine *ComputeEdge* is for computing the edge values of all pixels. The edge values of boundary pixels are exchanged in the routine *ExEdge*.

We use a typical image, that of a man body, as shown in Figure 7, to be the test pattern. The dots are 1-pixels, the white space is the 0-pixels, and the asterisks are the result of the thinning process. For the image of size 128×128, the number of iterations required by the

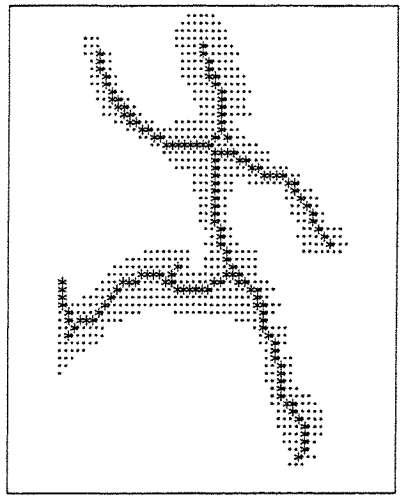

Figure 7: The image pattern and the thinning result

thinning algorithm is 15. The thinning time and other performance data of the algorithm for various numbers of the processors are tabularized in Table 3. The "efficiency" measure is to reflect the effectiveness of using more processors to solve the same problem. The loss of

Table 3: Performance of parallel thinning

Number of processors	1	2	3	4	5	6	7	8
Thinning time	53346	26987	19867	15230	12394	11148	9054	8679
Speedup	1	1.977	2.685	3.503	4.304	4.785	5.892	6.146
Efficiency	1	0.988	0.893	0.875	0.861	0.798	0.842	0.768
Communication cost (%)	—	1.112	1.510	1.970	2.318	2.691	3.313	3.457

the efficiency as the number of processors increases is due to interprocessor communication costs and load imbalances. From the thinning algorithm, we see that each thinning iteration step involves two communication operations with neighboring nodes (in a chain, each node has one or two neighbors): the exchange of boundary rows and the exchange of edge values of boundary pixels. An exchange operator is made up of sending and receiving a fixed size message in parallel. It is measured that the operator uses about 10 ticks. The total communication time is thus about 300 ($10 \times 2 \times 15$) ticks, which is the same for any number of processors. Its contribution in percentage to the overall thinning time is shown in the last row of Table 3.

We see that in parallel thinning, the computational requirement of a node is mainly dependent on the 1-pixels. The amount of conversions of 1-pixels to 0-pixels in an iteration step is unpredictable, and hence the computational workload can be somewhat varied over time. We thus resort to dynamic remapping to balance the workload over the course of thinning. We approximate the computational workload of a processor at an iteration by the processing time spent in the previous iteration. This approximation is reasonable since erosion takes place gradually along the edges of object skeletons, and thus the processing times of two consecutive iterations should not differ by a great deal.

From the experience in the WaTor simulation, we try two invocation policies for the remapping. One is to invoke the remapping once every two steps, and the other is to invoke the remapping only once at beginning of the thinning process. Since no computation time is available before the first iteration step to serve as an estimate of the workload, we perform the remapping between the first and the second iteration step. Figure 8 plots the processor utilization U_k across eight processors at various iteration steps for cases with and without remapping.

The curve for case 1 (without remapping) shows that the initial unbalanced workload distribution tends to uniform as the thinning algorithm proceeds. This points to the fact that the problem in question does not favor remapping. In fact, by applying the once-at-

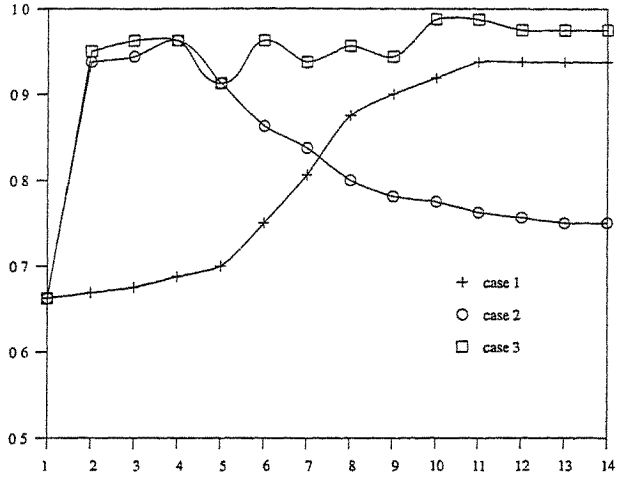

Figure 8: Processor utilizations at various iteration steps (8 processors)

the-beginning remapping to the problem (case 2), the overall performance seems to become worse: the initial balanced load distribution which is the result of the remapping, unfortunately, tends to non-uniform afterwards. To preserve the uniform distribution, it seems necessary to invoke the remapping periodically (case 3). This time improvement is evident, as can be seen in the figure. Then in Figure 9, we show the improvement due to remapping (cases 2 and 3) in overall thinning time for different numbers of processors. Note that even though case 2 has seemed to be not so satisfactory in terms of processor utilization as shown in Figure 8, it does however reap performance gain in terms of thinning time most of the time, even outperforming case 3 in some instances.

From Figure 9, it is clear that the parallel thinning algorithm does benefit from remapping. Although the once-at-the-beginning remapping could sometimes outperform frequent remapping, the latter tends to give smoother performance throughout. As we have already pointed out, this particular test image is unfavorable as far as remapping is concerned because its workload distribution would tend to a uniform distribution as thinning progresses. Therefore, we consider the saving due to remapping of only a few percents in the overall thinning time in both cases satisfactory. In comparison with the interprocessor communication cost which accounts also for a few percents (see Table 3), this saving is significant.

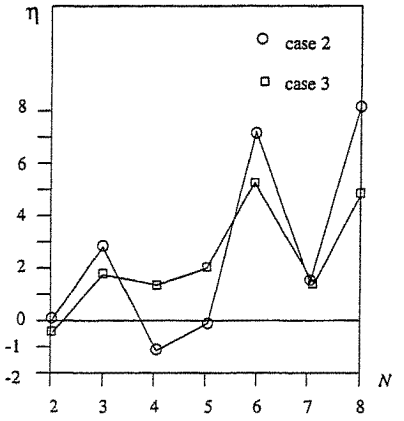

Figure 9: Improvement due to remapping for various numbers of processors.

5 Conclusion

In this paper, we study distributed remapping with the generalized dimension exchange method. We evaluate its performance in two data parallel computations. In the WaTor simulation of a 256×256 torodial ocean running on 16 processors, it is found that frequent remapping leads to about 15% improvement in simulation time over static mapping, and it outperforms centralized remapping by a factor of 50%. In parallel thinning of a 128×128 image on 8 processors, the policy of frequent remapping still saves about 5% thinning time although the test image is unsuitable for remapping. We consider these gains in performance (in the order of $10 - 20\%$) due to remapping satisfactory because our test problems are themselves balanced in statistical sense—*i.e.*, the mean and variance of the workload distribution are more or less homogeneous across the domain, and the imbalances are mainly due to statistical fluctuations. We believe this is typical of many real data parallel problems. For other problems that have substantial imbalances (the simulation of timed Petri nets, for instance), improvements on the order of hundreds of percents could sometimes be observed [17].

References

[1] D. C. Arney and J. E. Flaherty. An adaptive mesh-moving and local refinement method for time-dependent partial differential equations. *ACM Transactions on Mathematical Software*, 16(1):48–71, March 1990.

[2] M. J. Berger and S. H. Bohari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, May 1987.

[3] A. N. Choudhary, B. Narahari, and R. Krishnamurti. An efficient heuristic scheme for dynamic remapping of parallel computations. *Parallel Computing*, 19:621–632, 1993.

[4] A. N. Choudhary and R. Ponnusamy. Run-time data decomposition for parallel implementation of image processing and computer vision tasks. *Concurrency: Practice and Experience*, 4(4):313–334, June 1992.

[5] G. Cybenko. Load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[6] A. K. Dewdney. Computer recreations. *Science America*, December 1984.

[7] G. C. Fox. Applications of parallel supercomputers: Scientific results and computer science lessons. In *Natural and Artifical Parallel Computation*. The MIT press, Cambridge, 1990.

[8] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving problems on concurrent processors, Volume I.* Prentice Hall Inc., 1988.

[9] N. Francez. Distributed termination. *ACM Transactions on Programming Languages Systems*, 2:42–45, January 1980.

[10] R. V. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13(3):312–324, November 1991.

[11] S. Heydorn and P. Weidner. Optimization and performance analysis of thinning algorithm on parallel computers. *Parallel Computing*, 17:17–27, 1991.

[12] C. M. Holt, A. Stewart, M. Clint, and R. D. Perrott. An improved parallel thinning algorithm. *Communications of ACM*, 30(2):156–160, February 1987.

[13] S. H. Hosseini, B. Litow, M. Malkawi, J. Mcpherson, and K. Vairavan. Analysis of a graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing*, 10:160–166, 1990.

[14] J. De Keyser and D. Roose. A software tool for load balanced adaptive multiple grids on distributed memory computers. In *Proceedings of 6th Distributed Memory Computing Conference*, pages 122–128, April 1991.

[15] J. De Keyser and D. Roose. Multigrid with solution-adaptive irregular grids on distributed memory computers. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Parallel Computing*, pages 375–382. Elsevier Science Publishers, 1992.

[16] F. Mattern. Asynchronous distributed termination: parallel and symmetric solutions with echo algorithms. *Algorithmica*, pages 325–340, May 1990.

[17] D. M. Nicol. *Personal communications*, May 1993.

[18] D. M. Nicol and P. F. Reynolds. Optimal dynamic remapping of data parallel computation. *IEEE Trans. on Computers*, 39(2):206–219, February 1990.

[19] D. M. Nicol and J. H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9):1073–1087, September 1988.

[20] S. Ranka, Y. Won, and S. Sahni. Programming a hypercube multicomputer. *IEEE Software*, 5:69–77, September 1988.

[21] Y. Shih and J. Fier. Hypercube systems and key applications. In K. Hwang and D. Degroot, editors, *Parallel Processing for Supercomputers and Artifical Intelligence*, pages 203–243. McGraw-Hill Publishing Co., 1989.

[22] B. Szymanski, Y. Shi, and S. Prywes. Synchronized distributed termination. *IEEE Transactions on Software Engineering*, SE-11(10):1136–1140, October 1985.

[23] Inmos Limited (U.K.). *The Transputer Databook.* 1989.

[24] Inmos Limited (U.K.). *ANSI C Toolset User Manual.* 1990.

[25] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):451–481, October 1991.

[26] C.-Z. Xu and F.C.M. Lau. The generalized dimension exchange method for load balancing in k-ary n-cubes and variants. *Journal of Parallel and Distributed Computing.* To appear. (Available as Tech. Report TR-92-02, Dept. of Computer Science, The Univ. of Hong Kong, Jan. 1992).

[27] C.-Z. Xu and F.C.M. Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16(4):385–393, December 1992.

[28] C.-Z. Xu and F.C.M. Lau. Termination detection for loosely synchronized computations. In *Proceedings of 4th IEEE Symposium on Parallel and Distributed Processing*, pages 196–203, December 1992.