CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 9(12), 1351–1376 (DECEMBER 1997)

# Decentralized remapping of data parallel applications in distributed memory multiprocessors

CHENGZHONG XU[1], FRANCIS C. M. LAU[2] AND RALF DIEKMANN[3]

[1]*Dept. of Electrical & Computer Engineering, Wayne State University, Detroit, MI 48202*

[2]*Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong*

[3]*Department of Computer Science, University of Paderborn, Fuerstenallee 11, 33102 Paderborn, Germany*

**SUMMARY**

**In this paper we present a decentralized remapping method for data parallel applications on distributed memory multiprocessors. The method uses a generalized dimension exchange (GDE) algorithm periodically during the execution of an application to balance (remap) the system's workload. We implemented this remapping method in parallel WaTor simulations and parallel image thinning applications, and found it to be effective in reducing the computation time. The average performance gain is about** 20% **in the WaTor simulation of a** $256 \times 256$ **ocean grid on 16 processors, and up to** 8% **in the thinning of a typical image of size** $128 \times 128$ **on eight processors. The performance gains due to remapping in the image thinning case are reasonably substantial given the fact that the application by its very nature does not necessarily favor remapping. We also implemented this remapping method, using up to 32 processors, for partitioning and re-partitioning of grids in computational fluid dynamics. It was found that the GDE-based parallel refinement policy, coupled with simple geometric strategies, produces partitions that are comparable in quality to those from the best serial algorithms. ©1997 John Wiley & Sons, Ltd.**

## 1. INTRODUCTION

The mapping problem in parallel computations is concerned with how to distribute the workload or processes of a computation among the available processors so that each processor would have the same or nearly the same amount of work to do. In most cases, mapping 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[1]. For computations whose runtime behavior is non-deterministic or not so predictable, however, performing mapping only once in the beginning is insufficient. For these cases, it might be better to perform 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 load-balancing method, the *generalized dimension exchange* (GDE) method[2,3]. We demonstrate the effectiveness of this mechanism through incorporating it into the implementation of three major applications.

A data parallel computation decomposes its problem domain into a number of subdomains (data sets), and designates them to processes[1]. These processes simultaneously

CCC 1040–3108/97/121351–26 $17.50

*Received 27 July 1995*
*Revised 6 August 1996*

perform the same functions across different data sets. Because the sub-domains are connected at their boundaries, processes in neighboring sub-domains 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 computation 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[4].

In data parallel applications, 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. Such adaptive data parallel computations appear frequently in scientific and engineering applications such as those in molecular dynamics (MD) and computational fluid dynamics (CFD). A molecular dynamics program simulates the dynamic interactions among all atoms in a system of interest for a period of time. For each time step, the simulation calculates the forces between atoms, the energy of the whole structure and the movements of atoms. Since atoms tend to move around in the system, simulation loads associated with different parts of the system change from one step to another with the change of the atoms' spatial positions. A computational fluid dynamics program calculates the velocity and the pressure of vertices in a moving object for the purpose of deriving its structural and dynamic properties. The object can be either a car, an airplane, a space-shuttle or any other high-speed vehicle. It is first tessellated using a grid. Numerical calculations are carried out at grid points. In simulations that use adaptive grids to adjust the scale of resolution as the simulation progresses, computational workloads associated with different parts of a grid may change from phase to phase. To implement this kind of computation on a distributed memory multiprocessor, *static* domain decomposition techniques, such as strip-wise, box-wise and binary decompositions[5], are often not satisfactory; they fail to maintain an even distribution of computational workloads across the processors during execution. Because of the need for 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 1). 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 afresh – i.e. treating the current overall workload as if it is a 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 affairs. 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 systems. Therefore, the second approach of adjusting boundaries from the previous phase is preferred, which can be performed easily in a decentralized, parallel fashion. 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 adjust the sub-domain boundaries among processors (also in a distributed way) so that the result is a reasonably balanced workload.

This paper proposes a new and effective dynamic remapping method for time-varying 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 chance of running into bottleneck problems and is generally more reliable. The method requires no global information, and no broadcasting of information. Through actual implementations, the net gain in performance due to the use of this method is found to be 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 are based on knowledge of its nearest neighbors' states. GDE load-balancing is iterative: every processor successively balances its workload with every one of nearest neighbors in an iteration step until a global balance state is reached. This global balance state is detected by a distributed termination detection algorithm embedded in the load-balancing algorithm. We have analyzed the GDE method thoroughly in our previous works, and showed that it is effective, scalable and applicable to many network topologies[2,3]. This paper continues the study of the GDE method with an emphasis on its applicability to real problems, and also 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 4–6 evaluate the performance of the mechanism in three different data parallel applications. Section 7 summarizes related work in the literature. We conclude the paper in Section 8 with remarks on the GDE method on arbitrary computational graphs.

## 2. COMPUTATION MODEL AND THE GDE METHOD

### 2.1. Computation model

We consider time-varying data parallel computations in distributed memory multiprocessors. A distributed memory multiprocessor 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 $j$.

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 sub-domains of the problem[1]. 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 1 shows a typical scenario of the paradigm in a system of four processors. The horizontal scale corresponds to the 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 represent 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.

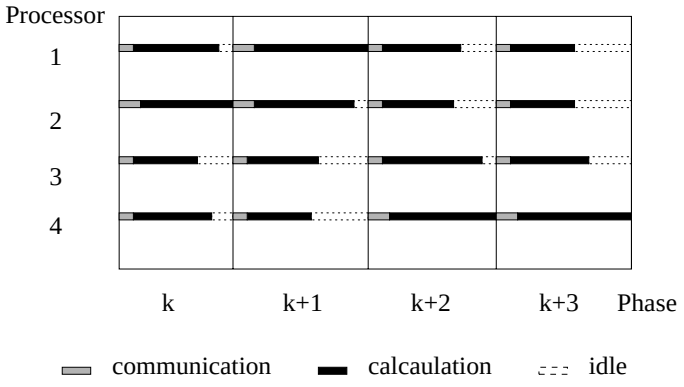


*Figure 1. An example of time-varying multiphase data parallel computations*

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 computation is dependent upon two main factors: the interprocessor communication cost and the degree of load imbalance 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 multiprocessor 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 Fox *et al.*[1] 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 a 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. The cost of remapping includes the cost of interprocessor communications and the cost of subsequent workload transfers. Note that the distribution of the computation times $t_{ik}$ has much bearing on the reward of remapping. Given some time-varying computations 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 phases to become severely imbalanced in the next phase. That is, for computations whose execution behavior is non-deterministic (and unpredictable), there exists the possibility that a version with no balancing would outperform a dynamically balanced version, 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 three we implemented.

### 2.2. The GDE method

Our remapping mechanism invokes a load-balancing procedure between every two successive phases, $k$ and $k+1$ say, so that hopefully the same 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 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 such that there are $t_i^{calc}$ pieces of work, each requiring one unit of execution time. In this 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. In the case that the communication time $t_i^{comm}$ is non-negligible, the remapping aims at balancing the calculation time of processors while retaining communication locality and maintaining the original interprocessor communication structure. It is done through nearest-neighbor shifting for the purpose of load-balancing.

Dynamic remapping incurs runtime overhead. To balance between the desire to balance the workloads of processors and the desire to minimize the remapping time, we have proposed several iterative methods for remapping[3,6]. These methods operate in a relaxation fashion in which a processor successively balances its workload with its nearest neighbors. Iterative methods have a less stringent requirement on the spread of local load information around the system. They are flexible in controlling the balance quality and suitable for retaining the communication locality.

The GDE method is one of the most efficient iterative nearest-neighbor methods.

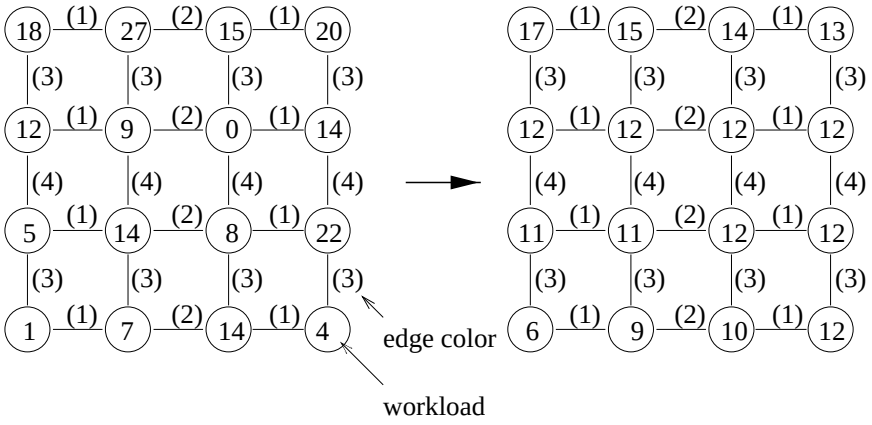


*Figure 2. Change of workload distribution after an iteration sweep in the GDE method*

It evolves from the dimension exchange (DE) method which was intensively studied in hypercube-structured multiprocessors (see [2] for a brief review). 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 is through a number of such iteration steps that the extra workload of a processor may get propagated to even the remotest processors. 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*)[7]. For non-hypercube structures, however, we proved that the DE method is not the most efficient (not optimal)[2]. 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 ($\kappa$, say), and no two adjoining edges are assigned the same color number. We index the colors with integers from 1 to $\kappa$. Figure 2(a) shows a colored $4 \times 4$ mesh. The numbers in parentheses are the assigned color numbers (or chromatic indices).

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 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* $\lambda$, $0 < \lambda < 1$. 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 $t_i$ ($t_j$) is the current workload of processor $i$ (processor $j$), 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[7,8]. Figure 2 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 $\lambda$ 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. It was proved that the performance of these optimally tuned procedures is scalable: the number of iteration steps required is found to be linearly proportional to $N$, the number of processors. 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 sufficiently small.

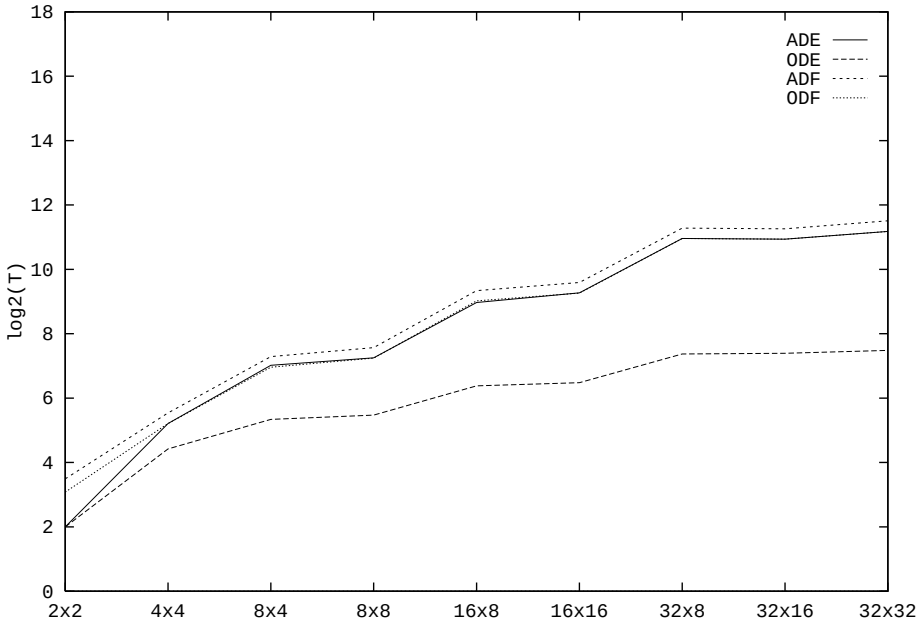


*Figure 3. The number of communication steps necessary for reaching a global balance state in a two-dimensional mesh of sizes varying from* $2 \times 2$ *to* $32 \times 32$ *(ADE represents the original dimension exchange method; ODE is the optimally-tuned GDE method; ADF represents a diffusion method; ODF is the optimally-tuned diffusion method)*

Figure 3 plots the number of communication steps necessary for reaching a global balance state from any initial workload distribution, using four different load-balancing algorithms. The diffusion method is another nearest-neighbor load-balancing algorithm, in which a processor balances its workload with those of its nearest neighbors simultaneously rather than one at a time as in the dimension exchange method. It is included in the

figure for comparison. This figure clearly indicates that the dimension exchange method outperforms the diffusion method, and that the optimized GDE method accelerates the dimension exchange load-balancing process significantly. From this figure it can also be seen that the number of communication steps in a two-dimensional mesh is dependent only on the size of its larger dimension and is insensitive to the size of its smaller dimension. This observation was proved to be true in both the mesh and the torus[3].

## 3. DISTRIBUTED REMAPPING WITH THE GDE METHOD

### 3.1. Distributed convergence detection

The distributed GDE load-balancing procedure is invoked between consecutive phases. Every time 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 areas. In the past, numerous solutions of diverse characteristics have been proposed. Readers are referred to [9] for a brief review. For our situation, we need a fully distributed and 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 [9], which is especially effective for termination detection of loosely synchronized iterative computations in general.

The method (see the algorithm below) makes use of virtual global 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, $\Delta$, which is a function of the structure of the underlying colored system graph. In [9], we showed that this $\Delta$ is in fact equal to the minimum number of *sweeps* required by a processor to acquire the status of other processors. In the color mesh of Figure 2(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 ≤ Δ) {
        for (c = 1; c ≤ κ; 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. Multithreaded implementation of the remapping mechanism

Even though the analysis of the GDE method ignores the interprocessor communication overhead, the method is applicable to the computations where interprocessor communication costs are non-negligible. Basically, what the GDE load-balancing ends up with are nearest-neighbor communications that shift the loads over 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 redecompositions.

The problem domain can be treated as a group of *internal load distributions* together with a corresponding *external load distribution*. Every sub-domain 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. 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 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 executed by the decision maker, which combines the GDE algorithm and the termination detection algorithm. Note that *LocalTerminated* would

become 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 Figure 2(a), we obtain the inflow or outflow value for each link, as illustrated in Figure 4.

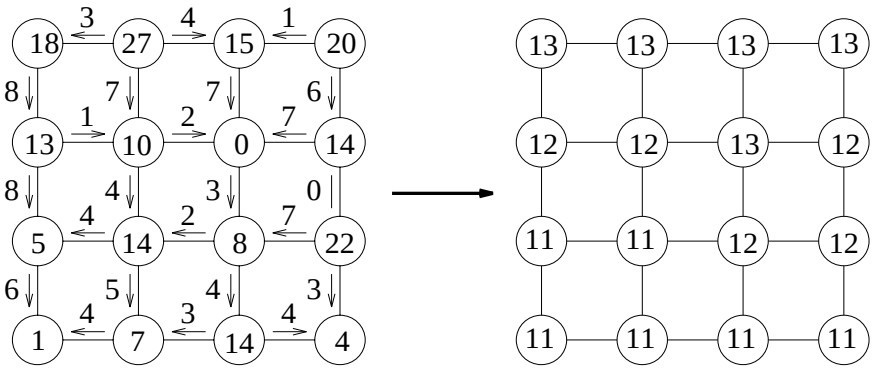


*Figure 4. In/out-flow along each channel of a processor necessary for arriving at a global balance state*

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 sub-domain. 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; what is done is basically adjusting of borders. For strip-wise-partitioned sub-domains, this can be done conveniently through shifting rows (or columns) between neighboring sub-domains. Details of how this is done are presented in Section 4 and 5 when the implementations of two real applications are discussed. In Section 6, we discuss a policy for splitting data items in grid partitioning and repartitioning in computational fluid dynamics.

Data transfer dominates the cost of remapping when the original distribution is severely imbalanced. Nearest-neighbor algorithms reduce the domain remapping operation to adjustment of sub-domain borders. Correspondingly, a processor may need to send and/or receive data along its communication channels at the same time. To exploit the parallelism in multiport communications, we experimented with a multithreaded model for multiport concurrent communications. We created a sender thread or receiver thread for each communication port. The local sub-domain is treated as a public data pool for all threads. Sender threads continually fetch data from the pool and receiver threads store data in the pool. One more thread was used for the pool management, whose tasks include the selection of data for sender threads and the combination of data from receiver threads with the original sub-domain.

The flow calculation determines the communication pattern and communication volume along each channel. A processor usually needs to send many data items to the same processor. These small pieces of data items destined for the same processor were aggregated into a single large message so as to reduce the number of instances of communication startup.

The time-varying data parallel applications we selected for testing the performance of the GDE-based remapping mechanism are the WaTor simulation[10] and parallel thinning of images[11,12]. 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. We also implemented the GDE-based remapping mechanism in parallel grid partitioning/repartitioning in the context of computational fluid dynamics. All experiments ran on a group of T805-30 transputers. The first two were coded in INMOS Parallel C[13], and the third in C under the Parix parallel operating system. The main metric in the first two experiments is the *improvement* in execution time due to remapping, denoted by $\eta_{remap}$, 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. The metric in the application of grid partitioning/repartitioning is the *improvement* in the quality of partitions due to refinement. Another metric for all experiments is the overhead of the remapping (repartitioning) mechanism.

## 4. APPLICATION 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 was divided into a fine grid which was 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 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. If it is mature with respect to the *minnow breeding age*, the minnow leaves a new minnow of age 0 in the original location.
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 locates a vacant position as 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 implemented the WaTor algorithm on a ring-structured system (a ring is a special case of a torus). The parallel implementation decomposed the ocean grid into strips, and assigned each of them to a processing node. Each simulation step consists of three sub-steps:

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 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 was done for a $256 \times 256$ ocean grid which was mapped onto 16 transputers; it was run for as many as 100 simulation steps. The *minnow breeding*, the *shark breeding* and the *shark starvation* parameters were set to be seven, 12, and five steps, respectively. Initially, the ocean was populated by minnows and sharks generated from a random uniform distribution, which were distributed by rows among the 16 transputers. The total simulation time for 100 steps was 17.58 s.

The computational requirement of a processor is proportional to the density of fishes in the strip that it is simulating. The more fishes exist, the more computation time is needed for the update. Owing to the tendency of the fishes to form schools dynamically and owing to their unpredictable behaviors, the processor utilization $U_k$ (as defined in Section 2) of

the processors does not vary quite so smoothly over time, as shown in the TIME curve of Figure 5. We applied remapping based on 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 used 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.

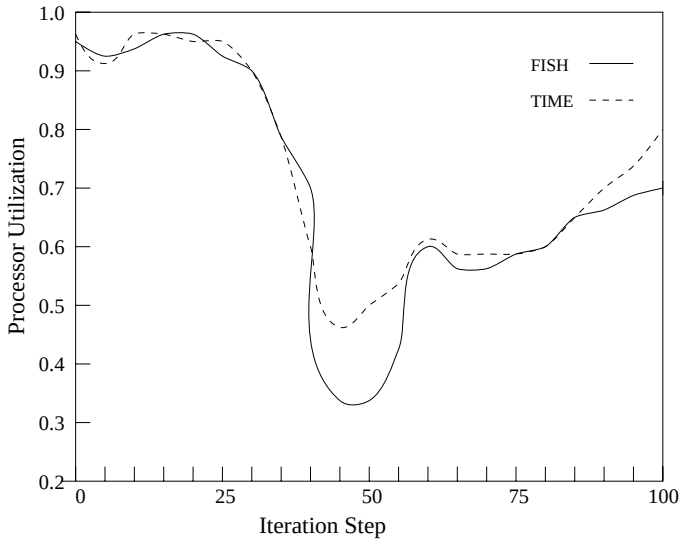


*Figure 5. Processor utilization at various simulation steps*

We examined 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 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 at the time the remapping is invoked and on the results of the decision making. Table 1 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 recorded 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 eight 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, $f_i^{post}$.

Table 1. 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 | 265.9 | | (-121, 0) | 37.3 | 1893 |
| 2 | 2094 | 285.7 | | (0,0) | 0.128 | 2094 |
| 3 | 1909 | 240.8 | | (0,0) | 0.128 | 1909 |
| 4 | 1927 | 254.3 | | (0,0) | 0.128 | 1927 |
| 5 | 2160 | 276.6 | | (0, -126) | 33.8 | 2034 |
| 6 | 2123 | 273.7 | | (126, -228) | 71.7 | 2021 |
| 7 | 1737 | 225.2 | | (228, 0) | 71.4 | 1965 |
| 8 | 1727 | 233.7 | (17, 2.75) | (0, 109) | 25.2 | 1836 |
| 9 | 2042 | 271.0 | | (-109, 0) | 25.2 | 1933 |
| 10 | 2132 | 268.4 | | (0, 0) | 0.128 | 2132 |
| 11 | 2017 | 255.8 | | (0, -117) | 34.9 | 1900 |
| 12 | 1876 | 264.4 | | (117, 0) | 34.8 | 1993 |
| 13 | 1743 | 231.2 | | (0, 0) | 0.128 | 1743 |
| 14 | 1848 | 251.1 | | (0, 122) | 30.7 | 1970 |
| 15 | 1931 | 249.4 | | (-122, 139) | 41.4 | 1948 |
| 16 | 2030 | 265.6 | | (-139, 121) | 41.3 | 2012 |

All time items $t_i$ in the table are in milliseconds.

It can be seen from the table that the time for decision making $t_i^{decision}$ (= 2.75 ms) is relatively insignificant when compared to the update time $t_i^{update}$ (122 ms 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.

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 30–50 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.

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. 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 64 $\mu$s and 128 $\mu$s, respectively.

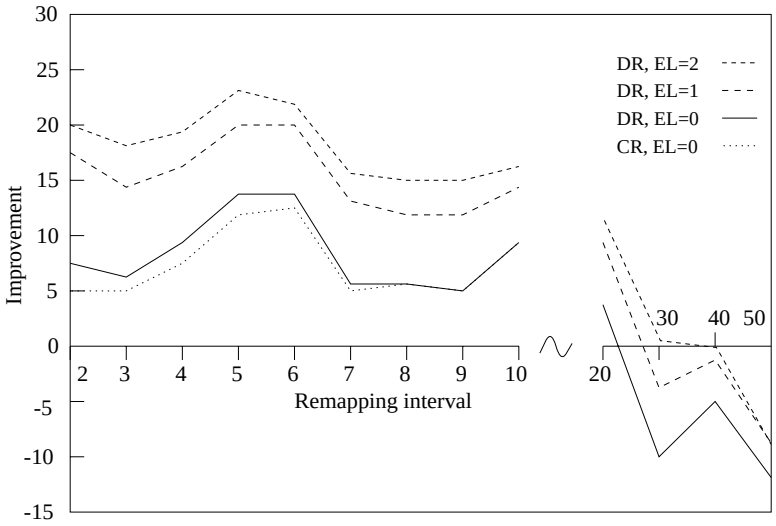


*Figure 6. Improvement due to remapping for various interval sizes*

For comparison, we also show the 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[14]. This centralized version takes a much longer time (6.4–12.8 ms) to do decision making than the decentralized version based on GDE. From the figure, it can be measured that the GDE-based remapping method, when frequently invoked, outperforms the centralized method by up to 40%.

## 5. APPLICATION 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, redundant information 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 picked an existing parallel

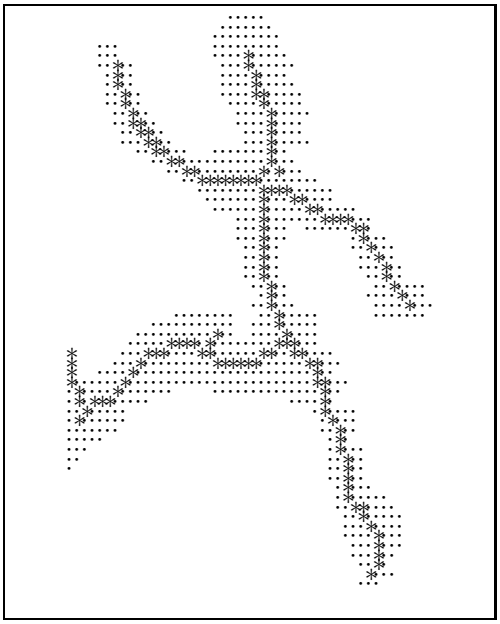

*Figure 7. The image pattern and the thinning result*

thinning algorithm, the HSCP algorithm[11], and implemented it on a chain-structured system using strip-wise 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.

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

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 used a typical image, that of a human 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 $128 \times 128$, the number of iterations required by the thinning algorithm is 15. The thinning time and other performance data of the algorithm for various numbers of the processors are given in Table 2. The 'efficiency' measure is to

Table 2. Performance of parallel thinning

| Number of processors | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Thinning time, s | 3.414 | 1.727 | 1.27 | 0.974 | 0.793 | 0.713 | 0.579 | 0.555 |
| 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 |

reflect the effectiveness of using more processors to solve the same problem. The loss of 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 0.64 ms. The total

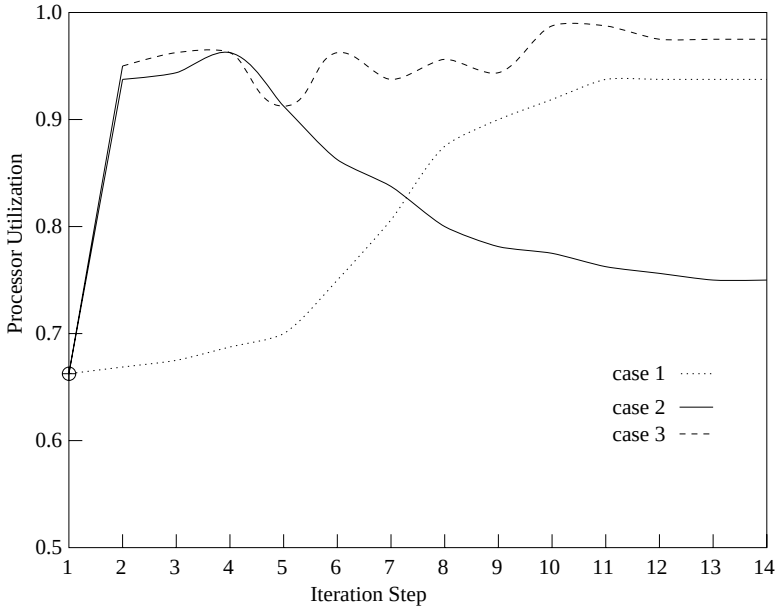


*Figure 8. Processor utilizations at various iteration steps (eight processors)*

communication time is thus $\sim$ 19.2 ms, 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 2.

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 a great deal.

From the experience of the WaTor simulation, we tried 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 the 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

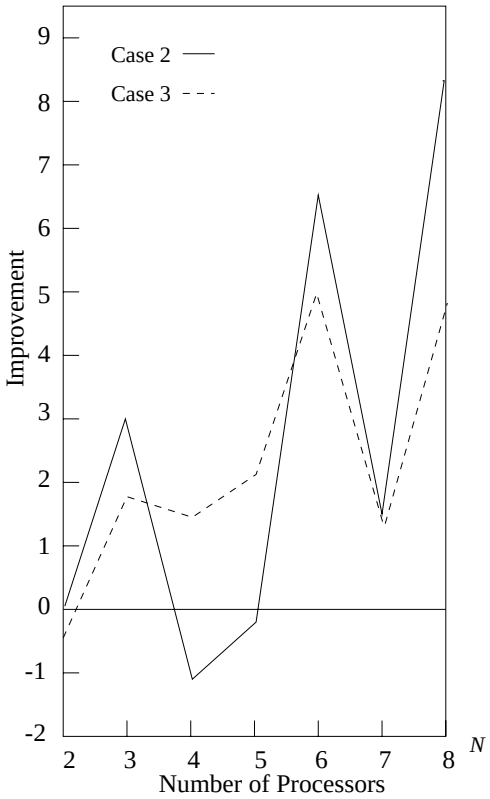


*Figure 9. Improvement due to remapping for various numbers of processors*

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-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 tends to be 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 not to be 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 percent in the overall thinning time to be satisfactory in both cases. In comparison with the interprocessor communication cost, which also accounts for a few percent (see Table 2), this saving is significant.

## 6. APPLICATION 3: PARALLEL UNSTRUCTURED GRID PARTITIONING

In computational fluid dynamics, physical domains of interest are tessellated using structured, block structured or unstructured grids. Unstructured grids, composed of triangular or tetrahedra elements, provide flexibility for tessellating about complex geometries and for adapting to flow features, such as shocks and boundary layers. Parallel numerical simulations require splitting the unstructured grid into equal-sized partitions so as to minimize the number of edges crossing partitions. Figure 10 shows an air-foil grid with 64 partitions.

Traditionally, grid partitioning is performed as a serial preprocessing step. For large unstructured grids, serial partitioning on a single processor may not be feasible due to memory or time constraints. Parallel partitioning is essential for reducing the preprocessing time. It is also desirable for handling load imbalances at runtime in simulations based on solution-adaptive grids.

We applied the GDE method to both grid partitioning and repartitioning. The grid in consideration was first partitioned using a simple strategy, and then assigned to processors using Bokhari's simple mapping algorithm[5]. We experimented with two initial partitioning strategies: the recursive orthogonal bisection (ROB) and Farhat's algorithm. The ROB algorithm recursively cuts the grid or sub-grids into two halves. It tends to generate balanced and well shaped partitions but with large cut size (the number of cut edges). Farhart's algorithm uses a breadth-first-search-based front technique to determine partitions one by one[15]. Starting from non-partitioned vertices which are connected to the boundary of partition $i-1$, it forms partition $i$ according to the number of edges external to partition $i-1$. The algorithm produces compact and balanced partitions with small cut size. Partitions generated by both strategies are not necessarily connected. We refined the distribution using the GDE method according to the geometric information of vertices with respect to the cut size.

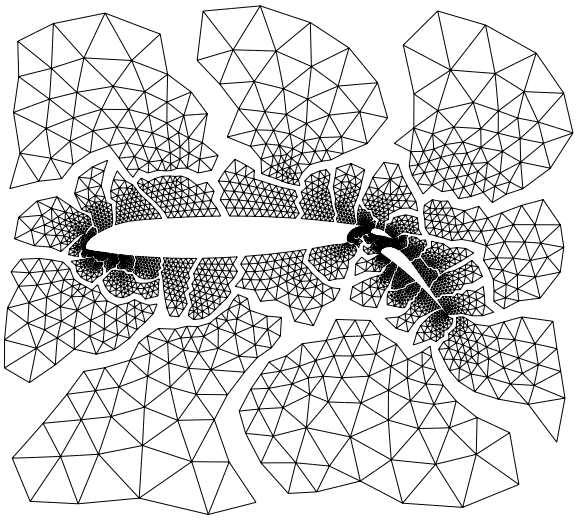

*Figure 10. A distribution of an unstructured mesh around an air-foil*

### 6.1. Flow calculation

Initial partitioning of a physical domain results in a computational graph, where vertices represent partitions and edges reflect neighboring relationships between partitions. The workload of a partition is defined as the total number of grid points inside the partition. The GDE method is readily applicable to the flow calculation in the case that the network graph matches well with the computation graph. In the case that the computational graph is different from the network graph, however, the GDE method on the network graph may generate a load flow along an edge between a pair of directly connected processors whose sub-domains are geometrically disconnected. Figure 11(a) shows such a computational graph mapped onto a $2 \times 4$ mesh. Network links $(2, 7)$ and $(3, 6)$ do not match with any edges of the computational graph. Load migration along such mis-matched links would lead to bad partition shapes and to non-connected partitions.

We handled this problem by derouting flows on mis-matched links to a sequence of edges that are on the shortest path to the destination of flows. In Figure 11(b), for example, the flow on link $(7, 2)$ is derouted to the path $(7, 3, 1)$ in the computational graph. The flow on link $(3, 6)$ is derouted to the path $(3, 5, 6)$. Notice that flow derouting may result in undesirable cyclic traffic such as those along the path $(7, 3, 5, 6, 7)$ in Figure 11(b). This cyclic path can be eliminated easily, as shown in Figure 11(c), by subtracting certain flows from all links in the path.

### 6.2. Selection of vertices for load migration

Selection of data sets to be split should be such that the geometric adjacency of data points in the problem domain is preserved. The fundamental idea behind the selection policy is the concept of gain associated with switching a grid point between different partitions. As in the Kernighan–Lin local refinement algorithm[16,17], the gain for a point $v$ was simply

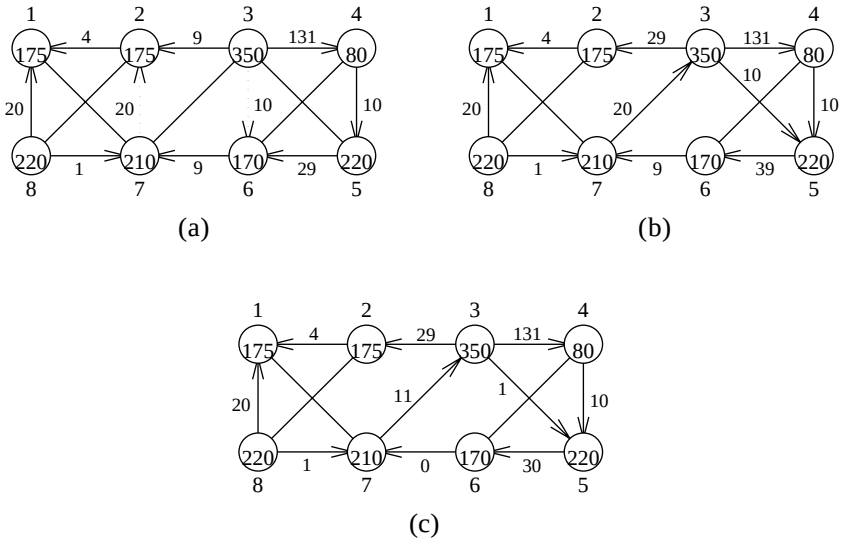


*Figure 11. (a) Mesh with mis-matched links; (b) flows along the mis-matched links derouted; (c) after elimination of cyclic traffic*

defined as the net reduction in the number of cut edges if the grid point were to switch sets. That is,

$$gain(v) = \sum_{(v,u)\ is\ an\ edge} \begin{cases} 1 & \text{if } P(u) = P(v) \\ -1 & \text{otherwise} \end{cases}$$

where $P(u)$ and $P(v)$ are the current partitions in which vertices $u$ and $v$, respectively, reside.

### 6.3. Experimental results

We experimented with the GDE-based refinement for grid partitioning[18]. Table 3 tabularizes the improvement in cut size due to GDE-based refinement for partitioning of three 2-D finite element unstructured grids onto eight or 32 processors. The airfoil grid has 4253 vertices and 12,289 edges, the crack grid has 10,240 vertices and 30,380 edges, and the big grid has 15,606 vertices and 45,878 edges[19]. From this table, it can be seen that the GDE-based refinement strategy improves the quality of initial partitions by a factor of 15–25% in most cases.

Table 4 presents the quality and efficiency of our parallel partitioning mechanism, together with those from other well-known sequential algorithms. The inertial method (IN) employs a physical analogy in which the grid points are treated as point masses and the grid is cut with a plane orthogonal to the principal inertial axis of the mass distribution. The spectral bisection method (SB) partitions a grid by considering an eigenvector of an associated matrix to gain an understanding of global properties of the grid. Both methods are complemented by a Kernighan–Lin (KL) local refinement algorithm. The multilevel method accelerates the spectral method by coarsening the grid down to a smaller graph. Partitions of the small graph resulted from the spectral method are projected back towards

Table 3. Improvements in cut size due to GDE-based refinement

| | Airfoil | | Crack | | Big | |
|---|---|---|---|---|---|---|
| | 8 | 32 | 8 | 32 | 8 | 32 |
| ROB | 272 | 870 | 556 | 1405 | 709 | 1950 |
| ROB+GDE | 207 | 666 | 463 | 1154 | 535 | 1478 |
| Improvement, % | 23.9 | 23.3 | 16.7 | 17.9 | 24.5 | 24.2 |
| Farhat | 423 | 684 | 611 | 1483 | 848 | 1454 |
| Farhat+GDE | 329 | 639 | 437 | 1146 | 719 | 1384 |
| Improvement, % | 22.2 | 6.6 | 28.0 | 22.7 | 15.2 | 4.8 |

the original grid. All these sequential methods are available in the Chaco library[20], and the evaluation was done on a Sun Sparc 10/50 with 96 MB memory. The parallel algorithm ran on a Parsytec T805 transputer-based GCel system. The T805 transputer is about 17 times slower than the Sparc machine; the timing performance in the table has been scaled to take this into account. From this table, it can be seen that the inertial method is fast, but produces partitions of relatively low quality. The spectral method produces excellent partitions at a very slow speed. The GDE-based refinement strategy, coupled with a simple Farhat partitioner, outperforms the inertial method in quality. It produces partitions comparable to the spectral method, and runs an order of magnitude faster.

Table 4. Comparison of various algorithms in terms of cut size and running time in seconds for mapping various unstructured meshes onto 16 processors (KL = Kernighan-Lin, IN = inertial, SB = spectral bisection, ML = multilevel and GDE = GDE-based refinement)

| | Airfoil | | Crack | | Big | |
|---|---|---|---|---|---|---|
| | CutSize | Time | CutSize | Time | CutSize | Time |
| IN | 503 | 0.20 | 797 | 0.45 | 1219 | 1.05 |
| IN+KL | 400 | 0.86 | 639 | 2.03 | 995 | 3.31 |
| SB | 372 | 21.29 | 671 | 74.14 | 863 | 124.8 |
| SB+KL | 309 | 24.20 | 577 | 82.60 | 675 | 131.3 |
| ML+KL | 325 | 2.98 | 615 | 6.47 | 605 | 8.05 |
| Farhat+GDE | 382 | 1.96 | 660 | 4.30 | 913 | 3.55 |

## 7. RELATED WORK

In the literature, much attention has been given to dynamic remapping of data parallel computations over the past several years. Nicol and Saltz addressed the issue of when to invoke a remapping so that its performance gain will not be offset by its overhead[21]. They proposed a simple heuristical invocation policy, Stop-At-Rise, for applications with gradually varying resource demands. Most recently, Moon and Saltz applied the Stop-At-Rise invocation policy, coupled with an elegant chain-structured partitioner and a recursive co-ordinate bisection (RCB) partitioner, to three-dimensional direct Monte Carlo simulation methods[22]. They showed that Stop-At-Rise remapping is superior to periodic remapping with any fixed intervals when the RCB is applied, and is slightly worse than periodic remapping with optimal intervals when the chain-structured partitioner is applied. A Stop-

At-Rise 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 repartitioning in the calculation of dynamic unstructured finite element grids[23,24]. The computation here 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 carrying out the remapping, the ROB, the simulated annealing, and the eigenvalue recursive bisection, and concluded that the latter should be preferred.

More parallel grid partitioning algorithms based on existing sequential algorithms have been developed in recent years. They include a parallel index-based algorithm[26], a parallel inertial algorithm[27], a parallel multilevel spectral algorithm[28], a parallel recursive bisection algorithm[29] and the unbalanced recursive bisection (a parallel variant of ROB)[30]. Most of these parallel algorithms take advantage of the recursive nature of their sequential versions, and exploit parallelism associated with the recursive step. They are efficient for initial grid partitioning. Since they are based on a global knowledge of the entire grid, they might not be applicable to grid repartitioning at runtime. Our GDE-based distributed refinement should serve as a complement to these parallel partitioning algorithms. In [27], the authors proposed a distributed greedy refinement strategy, based on ideas described by Hammond[19], to complement their parallel inertial algorithm. The refinement strategy greedily improves the bisection resulted from the inertial partitions in a pair-wise fashion. That is, all pairs of processors whose partitions share common edges exchange vertices so as to minimize cut sizes. Their parallel refinement strategy is essentially a dimension exchange algorithm applied to computational graphs.

In the context of image understanding, Choudhary *et al.* incorporated a remapping mechanism into a parallel motion estimation system[31,32]. 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 sub-domain it is working on to all the others, and then do border adjustment based on the collected information. A similar idea based on global knowledge was implemented by Hanxleden and Scott[14]. They invoked remapping periodically in the course of a Monte Carlo dynamical simulation, and gained 10–15% performance improvement using the optimal remapping interval.

## 8. CONCLUDING REMARKS

In this paper we have studied distributed remapping with the generalized dimension exchange (GDE) method for data parallel applications. We first evaluated its performance in two data parallel applications. In the WaTor simulation of a $256 \times 256$ torodial ocean running on 16 processors, it is found that frequent remapping leads to $\sim 20\%$ improvement in simulation time over static mapping, and it outperforms centralized remapping by a factor of 50%. In parallel thinning of a $128 \times 128$ image on eight processors, the policy of frequent remapping still saves $\sim 5\%$ thinning time, although the test image is unsuitable for remapping. We consider these gains in performance due to remapping to be satisfactory because our test problems are themselves balanced in the 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 that this characteristic is typical of many real data parallel problems. For other problems that have substantial imbalances (the simulation of timed Petri nets, for instance), improvements of the order of hundreds of percent could sometimes be observed.

We then implemented the GDE-based grid partitioning/repartitioning mechanism in the context of computational fluid dynamics. It is found that the GDE-based parallel refinement method, coupled with simple geometric approaches, produces partitions comparable in quality to results from the best serial algorithms.

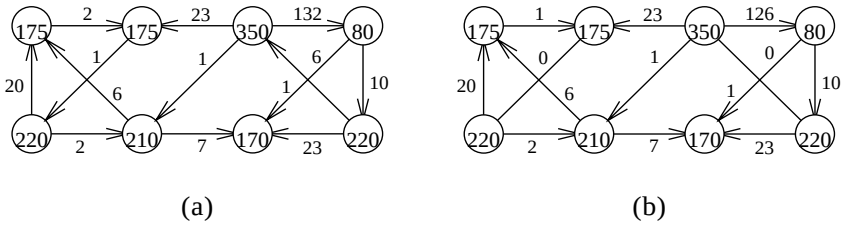


*Figure 12. Flows along each edge of a computational graph for arriving at a global balance state: (b) results from eliminating circular traffic paths in (a)*

The GDE method is readily applicable to the flow calculation in the case that the underlying network graph matches well with the computational graph generated from applications like the first two with which we experimented. In the case that the computational graph is different from the network graph, as we have encountered in the third experiment, we employed a derouting strategy to redirect traffic on mis-matched network links to a different path in the computational graph in order to preserve communication locality. An alternative approach is to apply the GDE method directly to the computational graph, as illustrated in Figure 12(a). Figure 12(b) is the result of eliminating the circular traffic flows in Figure 12(a). Flow calculation on the computational graph can also be performed in series.

## ACKNOWLEDGEMENTS

The authors would like to thank David Nicol for his helpful comments on an early version of this paper, Derk Meyer for his contributions to the third experiment, Bruce Hendrickson and Robert Leland for the Chaco library, and Steven Hammond for the example meshes. This work is partly supported by grants from the Research Grant Council of the Hong Kong Government, the University of Hong Kong, the DFG Sonderforschungsbereich 376 'Massive Parallelität', and the EC ESPRIT Long Term Research Project 20244 (ALCOM-IT). A preliminary version of this paper was presented at SHPCC'94, Tennessee, 23–25 May 1994[33].

## REFERENCES

1. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors, Vol. I*, Prentice-Hall, 1988.
2. C.-Z. Xu and F. C. M. Lau, 'Analysis of the generalized dimension exchange method for dynamic load balancing', *J. Parallel Distrib. Comput.*, **16**, (4), 385–393 (1992).

3. C.-Z. Xu and F. C. M. Lau, 'The generalized dimension exchange method for load balancing in $k$-ary $n$-cubes and variants', *J. Parallel Distrib. Comput.*, **24**, (1), 72–85 (1995).
4. G. C. Fox, 'Applications of parallel supercomputers: Scientific results and computer science lessons', in *Natural and Artifical Parallel Computation*, The MIT Press, Cambridge, MA, 1990.
5. M. J. Berger and S. H. Bohari, 'A partitioning strategy for nonuniform problems on multiprocessors', *IEEE Trans. Comput.*, **36**, (5), 570–580 (1987).
6. C.-Z. Xu, B. Monien, R. Lüling and F. C. M. Lau, 'Nearest neighbor algorithms for load balancing in parallel computers', *Concurrency: Pract. Exp.*, **7**, (7), 707–736 (1995).
7. G. Cybenko, 'Load balancing for distributed memory multiprocessors', *J. Parallel Distrib. Comput.*, **7**, 279–301 (1989).
8. S. H. Hosseini, B. Litow, M. Malkawi, J. Mcpherson and K. Vairavan, 'Analysis of a graph coloring based distributed load balancing algorithm', *J. Parallel Distrib. Comput.*, **10**, 160–166 (1990).
9. C.-Z. Xu and F. C. M. Lau, 'Efficient termination detection for loosely synchronous applications in multicomputers', *IEEE Trans. Parallel Distrib. Syst.*, **7**, (5), 537–544 (1996).
10. A. K. Dewdney, 'Computer recreations', *Sci. Am.*, **251**, (6), 14–22 (1984).
11. C. M. Holt, A. Stewart, M. Clint and R. D. Perrott, 'An improved parallel thinning algorithm', *Commun. ACM*, **30**, (2), 156–160 (1987).
12. S. Heydorn and P. Weidner, 'Optimization and performance analysis of thinning algorithm on parallel computers', *Parallel Computing*, **17**, 17–27 (1991).
13. Inmos Limited (UK), *ANSI C Toolset User Manual*, 1990.
14. R. V. Hanxleden and L. R. Scott, 'Load balancing on message passing architectures', *J. Parallel Distrib. Comput.*, **13**, (3), 312–324 (1991).
15. C. Farhat and H. D. Simon, 'TOP/DoMDEC—A software tool for mesh partitioning and parallel processing', *Technical Report RNR-93-011*, NASA Ames, 1993.
16. R. Diekmann, B. Monien and R. Preis, 'Using helpful sets to improve graph bisecitons', in Sotteau Hsu (Ed.), *Interconnection Networks and Mapping and Scheduling Parallel Computations*, DIMACS, 1995, pp. 57–73.
17. B. Kernighan and S. Lin, 'An efficient heuristic procedure for partitioning graphs', *Bell Syst. Tech. J.*, **29**, 291–307 (1970).
18. R. Diekmann, D. Meyer and B. Monien, 'Parallel decomposition of unstructured FEM-meshes', in *Proceedings of the 2nd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Springer LNCS 980, 1995, pp. 199–215.
19. H. W. Hammond, 'Mapping unstructured grid computations to massively parallel computers', PhD thesis, Rensselaer Polytechnic Institute, 1992.
20. B. Hendrickson and R. Leland, 'The chaco user's guide', *Technical Report SAND 93-2339*, Sandia National Lab., USA, 1993.
21. D. M. Nicol and J. H. Saltz, 'Dynamic remapping of parallel computations with varying resource demands', *IEEE Trans. Comput.*, **37**, (9), 1073–1087 (1988).
22. B. Moon and J. Saltz, 'Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures', in *Scalable High-Performance Computing Conference*, May 1994, pp. 176–183.
23. 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*, April 1991, pp. 122–128.
24. 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 (Eds.), *Parallel Computing*, Elsevier Science Publishers, 1992, pp. 375–382.
25. R. D. Williams, 'Performance of dynamic load balancing algorithms for unstructured mesh calculations', *Concurrency: Pract. Exp.*, **3**, (5), 451–481 (1991).
26. C.-W. Ou and S. Ranka, 'Parallel remapping algorithms for adaptive problems', *Technical Report SCCS-652*, School of Computer and Information Science, Syracuse University, 1995.
27. P. Diniz, S. Plimpton, B. Hendrickson and R. Leland, 'Parallel algorithms for dynamically partitioning unstructured grids', in *Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing*, 1995, pp. 615–620.

28. G. Karypis and V. Kumar, *Parallel multilevel graph partitioning*, Report, University of Minnesota, Department of Computer Science, May 1995.
29. C. Walshaw and M. Berzines, 'Dynamic load-balancing for pde solvers on adaptive unstructured meshes', *Concurrency: Pract. Exp.*, **7**, (1), 14–28 (1995).
30. M. T. Jones and P. E. Plassmann, 'Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes', in *Proceedings of 1994 Scalable High Performance Computing Conference*, IEEE Computer Society Press, May 1994, pp. 478–485.
31. A. N. Choudhary and R. Ponnusamy, 'Run-time data decomposition for parallel implementation of image processing and computer vision tasks', *Concurrency: Pract. Exp.*, **4**, (4), 313–334 (1992).
32. A. N. Choudhary, B. Narahari and R. Krishnamurti, 'An efficient heuristic scheme for dynamic remapping of parallel computations', *Parallel Comput.*, **19**, 621–632 (1993).
33. C.-Z. Xu and F. C. M. Lau, 'Decentralized remapping of data-parallel computations with the generalized dimension exchange method', in *Proceedings of 1994 Scalable High Performance Computing Conference*, IEEE Computer Society Press, May 1994, pp. 414–421.