Runtime Support for Parallelization of Data-Parallel Applications on Adaptive and Nonuniform Computational Environments

Maher Kaddoura and Sanjay Ranka
School of Computer and Information Science
4-116 Center for Science and Technology
Syracuse University
Syracuse, NY 13244-4100
kaddoura@top.cis.syr.edu ranka@top.cis.syr.edu

Preliminary Version

February 27, 1995

Abstract

In this paper we discuss the runtime support required for the parallelization of unstructured data-parallel applications on nonuniform and adaptive environments. The approach presented is reasonably general and is applicable to a wide variety of regular as well as irregular applications. We present performance results for the solution of an unstructured mesh on a cluster of heterogeneous workstations.

1 Introduction

Most computing environments consist of a cluster of nodes connected by a high-speed interconnection network. Node architectures include high-performance SIMD and MIMD parallel computers as well as numerous high-performance workstations. By pooling as many resources as possible, these environments represent the largest machine to which a researcher has access. This pool of resources may change over the lifetime of the computation due to machine failures or differing usage patterns. It should be possible to add or remove computational resources without significantly affecting the other machines and without changing the existing software. In such an environment an individual machine can be dedicated to a single user's computation or shared by users. The former has the advantage of providing static computing capability for each machine, while the latter has a higher rate of utilization. The resources available to the user may be classified as:

1. Static: Computational resources are fixed throughout the completion of all tasks.
2. Dynamic: Computational resources vary dynamically throughout the computation because of sharing among users.
3. Adaptive: Computational resources remain fixed for a reasonable interval of time followed by a change.

Efficient parallelization of data-parallel applications require careful attention to:

- Load Balance: The computational load on each processor should be proportional to the processor's computational power.
- Data Partitioning: Data should be partitioned such that nonlocal data accesses are minimized. This results in low communication costs.

Several methods of data partitioning to achieve efficient parallelization of data-parallel applications for static computational environments have been discussed in the literature and are part of data-parallel languages such as High-Performance Fortran [17] and potential extensions [16].

Limited research has been targeted towards parallel compilers and runtime support for nonuniform and/or adaptive environments. Nedeljkovic and Quinn [23] developed a data-parallel C compiler with dynamic load balancing for a network of workstations. Siegell and Steenkiste [29] implemented a runtime system that supports automatically generated programs with dynamic load balancing for workstations. Keyser, Lust, and Roose [22] implemented a parallel 2-D multiblock Euler/Navier-Stokes solver with adaptive block refinement and runtime load balancing for different parallel architecture, including clusters of workstations.

In this paper we discuss the runtime support required for the parallelization of unstructured mesh on a cluster of workstations. Many of these optimizations and issues are equally important for parallelization of a wide variety of structured as well as unstructured applications on an adaptive computing environment. The software developed is part of the STANCE (Software Techniques for Adaptive and Nonuniform Computational Environments) runtime library [18].

The remainder of the paper is organized as follows. Section 2 discusses the computational environment and the important issues and major contributions of this research. Section 3 describes the runtime support library. Section 4 presents performance measures for nonuniform and adaptive environments. Section 5 presents the performance of the library on a cluster of heterogeneous workstations connected by Ethernet. We conclude in Section 6.

2 Computational environment

Our model is restricted to the Single Processor Multiple Data (SPMD) model of execution. In this model the same program is executed on all processors. Parallelism is achieved by partitioning the data structures and associated computations among processors. We are targeting a nonuniform computational environment where the computational resources available may change adaptively.

- These changes should be gradual enough that remapping is not required as soon as the computational resources adapt. Data-parallel programs execute by iterating through a sequence of several phases. There is an implicit synchronization at the end of execution of every phase. We assume that remapping can be performed after a phase is completed. The effect of the change in computational resources during the execution of one phase is not expected to cause the overall performance to deteriorate significantly.

- Minimal amount of computational resources are available for the remapping and redistribution of data. Clearly, one can terminate the process as soon as it stops performing effective computation for the given data-parallel application. However, when the resource is available again this may require spawning a new process that may be considerably more expensive.

It is currently left to the programmer to choose the specific places in the program where checks are made to ensure that the effects of any change of available computational resources warrant a redistribution of the data.

Important issues and contributions

In the following we describe the important issues for the parallelization of unstructured data-parallel applications on adaptive environments:

1. **Fast Methods for Remapping** The amount of available computational resources may change during computation, which may require redistributing data items to achieve load balancing. It is important that this redistribution be done such that locality is maintained after the redistribution. Most unstructured data-parallel applications can be represented as computational graphs. We use a simple architecture-independent transformation that permutes all the nodes of the graph such that locality is improved. Let $T : V \longrightarrow \{1, 2, 3, \ldots, n\}$ define the above permutation. The goal of this transformation is to achieve good partitioning for a wide range of partitions. Several methods for achieving this transformation are described in

[7, 19] and elaborated on in Section 3. Mapping and remapping becomes relatively easy once this transformation is available.

2. **Minimization of Communication Cost** Several optimizations can be performed to reduce the amount of communication, including the removal of duplicate accesses and message coalescing [27]. For many data-parallel applications the accesses are symmetric. We describe in Section 3 several methods to reduce communication requirements for such cases in.

3. **Minimization of Redistribution Cost** There are several good ways to repartition data. The communication cost of redistribution can be reduced by choosing a repartitioning that minimizes the amount of data movement among the processors. We describe several strategies in Section 3.

4. **Address Translation** Parallel loops can be transformed into an inspector and an executor [27]. The inspector examines the data references and computes the off-processor data to be fetched. It also computes where the data will be stored once it is received. The executor uses this information to perform its computation.

   The use of a one-dimensional representation removes the necessity for maintaining explicit translation tables. The only information required at every node is the current partitioning of a one-dimensional list (memory requirements are proportional to the number of processors). This can be used to locally determine the location of all the data items.

3 Runtime support

Phase A	Data Partitioning	Transforms a graph into one-dimensional list
Phase B	Inspector	Translates indices; generates schedules
Phase C	Executor	Uses schedules for data movement; executes computations
Phase D	Load Balancing	Monitors load on each processor; redistributes data

Figure 1: The four phases required for parallelization iterative unstructured applications

Parallelization of iterative and unstructured data-parallel applications requires four major phases (see Figure 1). The first phase involves data partitioning. In this phase the nodes of the graph are renumbered to improve locality, which makes it easy to repartition the graph when the available resources change. The next two phases concern analyzing data-access patterns and communication between processors. The last phase involves load balancing, in which the load on each processor is monitored and, if necessary, the data is redistributed to balance the load. In static environments phase C tends to be executed multiple times, while phase B is executed once. In adaptive environments and/or adaptive applications[1] phase B is executed whenever data is redistributed.

[1] For these classes of applications the computational structure adapts after every few iterations.

3.1 One-Dimensional model of locality

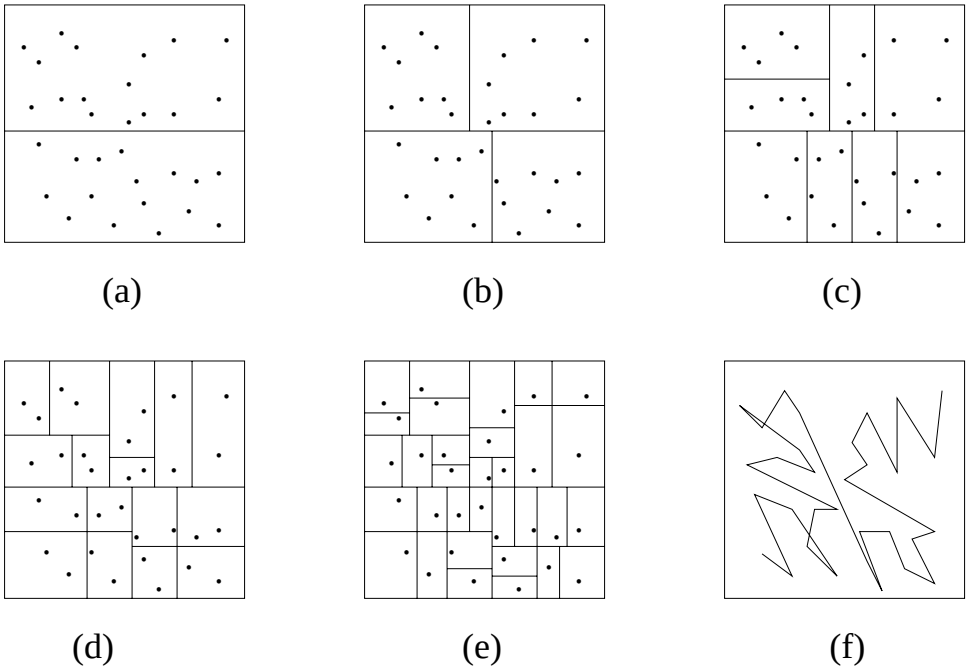


Figure 2: Mapping a graph into one-dimensional space using recursive coordinate bisection

A large number of unstructured data-parallel applications [8] can be represented as computational graphs from the perspective of parallel computing. The nodes of these graphs represent tasks that can be executed concurrently, while the edges represent the interactions between them. Further, the computational graphs derived from many applications are such that the vertices correspond to two- or three-dimensional coordinates, and the interaction between computations is limited to vertices that are physically proximate.

Several graph-partitioning methods are described in the literature. There are simple and fast heuristics for achieving partitioning by clustering physically proximate nodes (based on coordinate information) in two or three dimensions. Important heuristics include recursive coordinate bisection, inertial bisection, scattered decomposition, geometry-based partitioners, and index-based partitioners [9, 12, 13, 6, 25, 30, 32]. There are a number of methods that use explicit edge information to achieve better partitioning. Important heuristics include simulated annealing, mean-field annealing, recursive spectral bisection, recursive spectral multisection, mincut-based methods, and genetic algorithms [1, 11, 10, 14, 15, 21, 20, 26].

When computational resources are nonuniform, the parallelization of this computational graph requires partitioning the graph such that each processor is assigned nodes with computational weight proportional to the computational capabilities of that processor, and the number of cross edges are minimized. In adaptive environments there is a need to remap the graph when the available computational resources adapt according to the new computational capabilities of the processors. Many of the above methods are computationally expensive and thus are not suitable

for such environments.

We have shown that computational graphs representing applications from the physical domain (i.e., embedded in two or three dimensions) can be transformed into a simple architecture-independent one-dimensional representation that encapsulates the locality in these graphs (see Figure 2). This representation allows for a fast mapping of the computational graph onto the underlying computational resources at the time of execution. Let the nodes of the vertex set be numbered from 1 through n. The architecture-independent transformation permutes all the nodes of the graph such that locality is improved. Let $T : V \longrightarrow \{1, 2, 3, \ldots, n\}$ define the above permutation. The goal of this transformation is to achieve good partitioning for a wide range of partitions. Several methods for achieving this transformation are described in [19, 7]. After the initial transformation it is inexpensive to partition the one-dimensional list among the processors according to their computational capability, since partitioning is equivalent to assigning contiguous blocks of vertices to each partition. The size of each block is proportional to the weight of the partition. When the computational resources adapt, the same transformation can be used for repartitioning. Several algorithms for achieving this transformation and their performance are described in [19].

3.2 Inspector

In this section we outline the preprocessing needed by the inspector to generate the arguments required by the executor to perform the computations. The inspector has two main functions: data referencing, and generating a communication schedule [27].

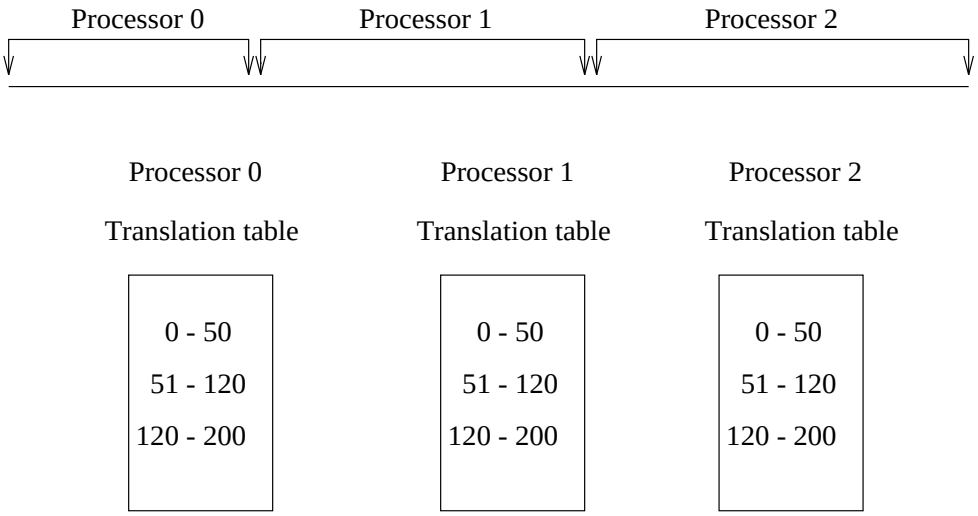


Figure 3: Translation table

Data Referencing The library supports a translation mechanism using a *translation table*. A simple implementation of a translation table stores, for each element, the name of its home processor and its local address in its home processor [27]. *Dereferencing* an element converts a global index

into a (processor, local index) tuple using the translation table. The translation table may be replicated among the processors or distributed among the processors using a fixed distribution (e.g., block or cyclic). When the table is replicated, dereferencing does not require communication; dereferencing may require communication when this table is distributed. Due to high memory requirements, replicating the translation table is not feasible for applications with large data sets.

When a one-dimensional transformation is used (Section 3.1), each processor is assigned an interval of data elements. Storing the first and last elements belonging to every processor in the transformed space is sufficient to generate the (processor, local index) tuple. The size of this list is proportional to the number of processors. It can be replicated on each processor (see Figure 3). To find the home processor of a particular element the list is searched until the processor holding the element is found. A processor holds an element if the element is greater than or equal to the first element that belongs to the processor, and less than or equal to the last element that belongs to it. The local address of a particular element is computed by subtracting it from the first element that belongs to its home processor. Although the computation cost of the translation using this table is significant, it is negligible compared to the cost of using communication for dereferencing using the simple scheme.

Communication Schedules Communication schedules are used to fetch nonlocal data elements into a local buffer or/and to scatter local data elements to other processors. Each processor provides the following information to generate a communication schedule:

1. Local list: local references to be gathered from or scattered to other processors
2. Processor list: processors to be gathered from or scattered to
3. Data size: Size of data elements involved in the gathering or scattering

The following information is available at a given processor P at this stage:

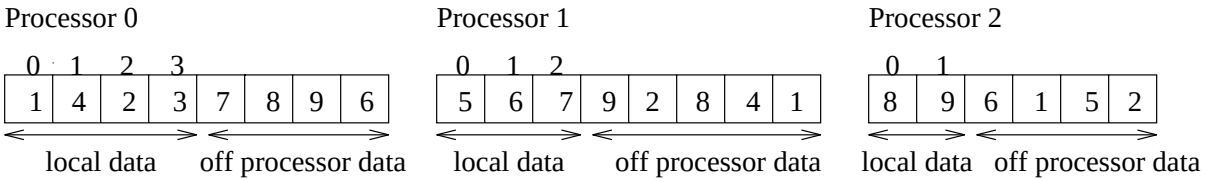
1. Send list: a list of arrays that store the local references of processor P that must be sent to other processors. The size of each array is maintained.
2. Permutation list: an array that stores the placement order in the local buffer of P for the data elements that processor P will receive when the schedule is used in the executor phase. It also includes information about the sizes of the messages that P will receive from other processors.

Efficient generation of communication schedules for nonlocal references can be done using two phases. The first phase removes duplicate accesses to avoid fetching a data item more then once. This is done by using a hash table [27]. The global references of the unique data elements are changed to local references in the hash table. Using the translation table, a communication schedule is created for accessing nonlocal accesses. This requires sending to the destination processor(s) a list of the different accesses that are required.

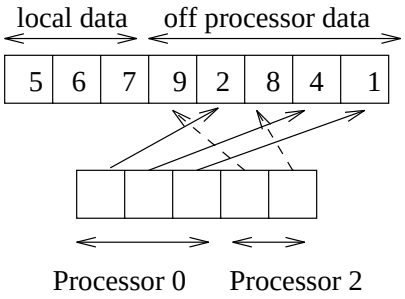
Data distribution

Processor 0	Processor 1	Processor 2
1: 7, 8	5: 6, 5, 9	8: 6, 1
4: 7, 2	6: 5, 2, 8	9: 5, 2
2: 4, 3, 9, 6	7: 4, 1	
3: 1, 2		

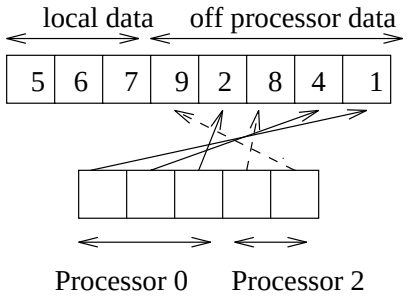
Local references of data in their home processors



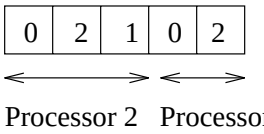
Permutation list of processor 1 before sorting



Permutation list of processor 1 after sorting each segment according to the local references of the nodes



Send list of processor 0 before sorting



Send list of processor 0 after sorting

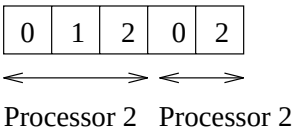


Figure 4: Schedule_sort1

For many irregular applications the accesses are symmetric (commutative) in nature (i.e., iterative techniques for the finite element method). If nodes n_1 and n_2 are stored on different processors and there is an edge between them, then the processor that stores n_1 will access n_2 and vice versa. One can exploit this symmetry to eliminate the communication required to generate the communication schedule. Although a processor may be able to determine the nodes it needs to send to every processor, it will not be able to determine the order in which these nodes are sent. Sorting of nodes based on their indices can determine the correct order of the nodes. This optimization is useful only when the cost of sorting is much smaller than the cost of off-processor accesses.

We have developed two methods for building communication schedules based on the above optimizations. We shall refer to them as *schedule_sort1* and *schedule_sort2*. In schedule_sort1 we sort both the sending list and the permutation list of each processor in increasing order. Each segment of the permutation list which points to the locations of the nodes that will be received from a particular processor is sorted according to the local references of these nodes in their home processor. Each segment of the sending list is sorted independently, thus the contents of each message is sent in increasing order and received in the same order (see Figure 4). Sorting the sending list can be avoided if a restriction is added that the nodes are traversed in increasing order according to their local references when building a communication schedule. We shall refer to this method as *schedule_sort2*.

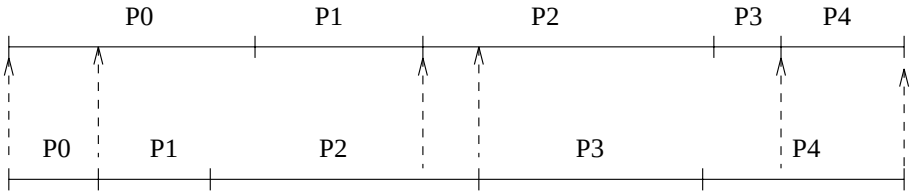
3.3 Executor

The executor uses the communication schedules generated by the inspector to move data between the processors in the environments and to perform the necessary computations. There are two basic primitives, *gather* and *scatter*. *Gather* is used to fetch off-processor elements, while *scatter* is used to to send off-processor elements.

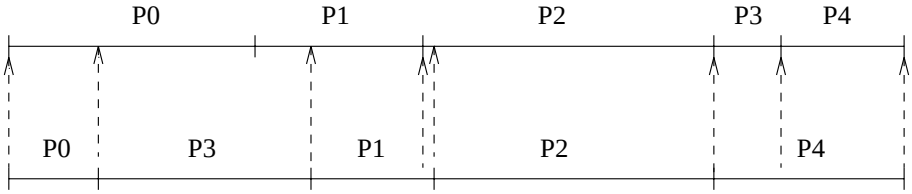
3.4 Minimizing the amount of data movement

There are several ways to achieve the repartitioning such that contiguous blocks are assigned to every processor. We will use the term *arrangements* to represent each of the possible ways of partitioning. There are $p!$ arrangements for p processors. We discuss a simple strategy for the minimization of the communication cost of redistributing data items. The two factors contributing to data redistribution time are the amount of data to be transferred and the number of messages generated.

The amount of data movement can be reduced by finding a new arrangement that maximizes the overlap between the original intervals and the new intervals. For example, consider a list of 100 elements and 5 processors with the following ratios of computational capabilities: $P_0 = 0.27, P_1 = 0.18, P_2 = 0.34, P_3 = 0.07$, and $P_4 = 0.14$. Let us assume that the one-dimensional list is divided among the processors using the arrangement $(P_0, P_1, P_2, P_3, P_4)$. If the computational capabilities of the processors adapts to 0.10, 0.13, 0.29, 0.24, 0.24, respectively, then dividing the list according to the original arrangement $(P_0, P_1, P_2, P_3, P_4)$ will yield 29 overlapped elements (see Figure 5 (a)) (i.e., 71 elements have to be moved across the network). On the other hand, if the



(a)



(b)

Figure 5: Different ways of repartitioning data items

---

procedure $MinimizeCostRedistribution(LIST, p, LIST\_OUT)$

/* p is the number of processors.
$LIST$ is the array which has the arrangement of the processors.
The function $COST$ given two different arrangements
of processors returns the cost of data redistribution.
$LIST\_OUT$ is the array which contains the arrangement of
processors generated by the procedure
*/

```
    for(1 ≤ i ≤ p)LIST_OUT[i] := LIST[i]
     max := −1. jmax := −1.
    for(1 ≤ i ≤ p)
        for(1 ≤ j ≤ p)
                MOVE(LIST_OUT, LIST[i], j).
                temp := COST(LIST, LIST_OUT).
                if (temp > max)
                    max := temp. jmax := j.
         MOVE(LIST_OUT, LIST[i], jmax).
end.
```

Figure 6: MinimizeCostRedistribution Algorithm

---

procedure $MOVE(LIST, C, L)$

/* Move the element C in $LIST$ (current arrangment of processors) in location L and rearrange the remaining elemnts. */
/* $MOVE(\{1,3,5,4,6\},5,0) = \{5,1,3,4,6\}$ */

find the location of C in $LIST$. We shall refer to this
location as X.
if $(X < L)$
shift the elements in location $X+1$ to L to the left.
if $(X > L)$
shift the elements in location $X+1$ to L to the right
put C in location L.
end.

Figure 7: Rearranging a list

list is divided using the arrangement $(P_0, P_3, P_1, P_2, P_4)$, the number of overlapped elements will increase to 65 (see Figure 5 (b)). The number of messages generated can also be taken into account by incorporating it into the cost of redistribution. Using the first arrangement (Figure 5 (a)), the number of messages needed to redistribute the data is 5; the number of messages needed to redistribute the data for the latter arrangement (Figure 5 (b)) is only 3.

Choosing the best arrangement by trying out all cases is feasible only for a small number of processors. Figure 6 gives a simple greedy algorithm which generates only a subset of all the arrangements, considering data overlap and number of messages generated. Our simulations show that this algorithm (MinimizeCostRedistribution (MCR)) produces good suboptimal results. The algorithm MOVE, which is used by MCR, is described in Figure 7. The time requirement for this algorithm is $O(p^3)$, where p is the number of processors.

3.5 Adaptive load balancing

When the available computational resources adapt, a remapping of data items may be required to maintain good load balance. This can be divided into four phases:

- Monitoring local load on each processor.
- Exchanging load information between processors.
- Making a decision to remap; if remapping is required, choosing the appropriate partitioning of the array to minimize data movement.
- If remapping is required, performing the data movement.

In our current implementation each processor monitors its own load and sends it to a *controller* processor, which makes the decision about repartitioning the data. Centralized load-balancing

algorithms are suitable for an environment with a small number of processors. This currently requires sending the load information as separate messages to the controller, which broadcasts the decision to all the processors. When better resource management tools are available, we hope to have distributed strategies.

The goal of a good parallelization for the targeted environment is to minimize the idle time on any given processor. Using information from the current phase, the data (and associated computations) should be redistributed such that the idle time for the next phase is minimized. This assumes that the computational resources allocated for the data parallel computation are the same as for the previous phase.[2] The controller determines from time to time whether the remapping of data is profitable. Remapping is considered profitable if its cost is offset by an improvement in time for the next phase. If it is not profitable, the controller broadcasts an appropriate message to all the processors, and computations are resumed for the next phase. Otherwise, the controller computes new data intervals for each processor based on its estimated computational capability in the previous phase. The new intervals are broadcast to all the processors and the data is redistributed among the processors.

The frequency of this load-balancing check has to be set based on the following:

- The overhead of load balancing. This should represent a small fraction of the time between successive load-balancing steps
- The rate at which the underlying computational resources adapt. If the computational environment adapts slowly, the frequency can be low. Clearly, if the computational resources adapt very frequently, effective parallelization will not be possible.

Techniques to choose the best frequency are outside the scope of this paper.

The controller receives the new computational capability of the processors and determines whether remapping the data is profitable. Remapping is considered profitable if the effect of the change in the load is expected to improve the overall computation time for the environment in the next phase to offset the cost of remapping. If remapping is not profitable, the controller processor broadcasts an appropriate message to the processors and computations are resumed for the next iteration.

3.6 Other communication optimizations

Latency is an important factor when performing parallel computing on a general network. The number of messages generated by our library could be reduced significantly by using multicast. Our library has the ability to use multicast to perform all communications between processors in the environments if the network supports multicast (e.g., Ethernet [3], ATM [2]).

[2]This could be extended to techniques that would predict the available computational resources based on more than one previous phase. If the operating system can guarantee that a process will be allocated a particular amount of resources for the next phase, this can also be used to predict the amount of computational resources available in the next phase.

4 Performance measures

The performance of a parallel application is usually measured in terms of speedup and efficiency. It is difficult to have analagous terms for nonuniform computational environment. In this section we give a general definition of efficiency that is suitable for data-parallel applications in a nonuniform environment. Let the amount of time required for computing a task be given by $T(p_i)$ on processors i if it is executed sequentially. Thus, processors i can complete $\frac{1}{T(p_i)}$ of the task per unit time. Collectively all the processors can complete (assuming no parallelization overheads) $\sum_{i=1}^{n} \frac{1}{T(p_i)}$ of the task per unit time. Thus, one can define the efficiency of parallelization as

$$E(p_1, p_2, \ldots, p_n) = \frac{\frac{1}{T(p_1, p_2, \ldots, p_n)}}{\sum_{i=1}^{n} \frac{1}{T(p_i)}} ,$$

where $T(p_1, p_2, \ldots, p_n)$ represents the time taken for completing the task when processors $p_1, p_2, \ldots, p_n$ are all used together.

For adaptive computational environments, assume that $T(p_1, p_2, \ldots, p_n)$ is the total time taken for completing the task. Let the fraction of the whole task which could have been completed by processor i during that time be given by $f_i(T)$. Then the efficiency of the parallelization can be given by

$$E(p_1, p_2, \ldots, p_n) = \frac{1}{\sum_{i=1}^{n} f_i(T)} .$$

Unfortunately, the value of $f_i(T)$ is difficult to compute in an adaptive environment.

5 Experimental results

In this section we study the effectiveness of the different optimizations suggested in the previous section. We evaluated the library on a cluster of SUN4 workstations connected by Ethernet using the P4 message-passing environment.

Workstations	Time
3	0.00033
5	0.00049
10	0.0025
15	0.0074
20	0.017

Table 1: Execution time of MinimizeCostRedistribution (in seconds)

Table 1 shows the execution time of MinimizeCostRedistribution in seconds. Its execution time is small, even for 20 processors. Table 2 shows the average cost of remapping different array sizes (floating point) over 100 randomly generated samples. These results show that using the

heuristic improved the cost of data remapping in all cases. It also shows that the total time required for remapping (with or without the optimization) is very small. This is critical for effective parallelization.

```
real y(number of vertices), t(number of vertices)   /* data arrays */
integer ia(number of edges)                          /* indirection array */

  k := 0.
  for(1 ≤ i ≤ number of vertices)
    t[i] := 0.
    for(1 ≤ j ≤ number of vertices connected to i)
      k := k + 1.
      t[i] := t[i] + y(ia(k)).
  for(1 ≤ i ≤ number of vertices)
    y[i] := t[i]/ number of vertices connected to i.
```

Figure 8: Irregular loop to be parallelized

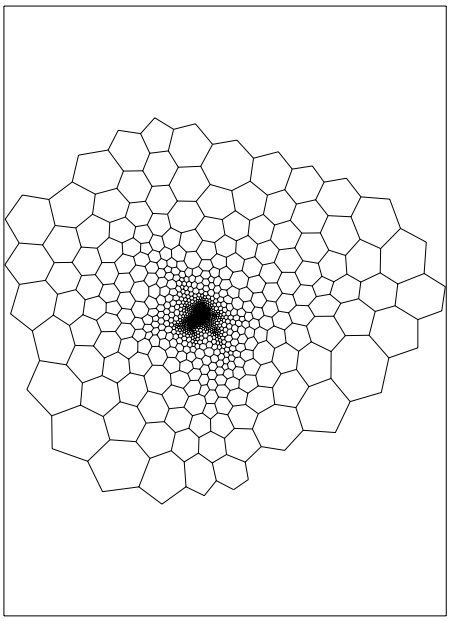


Figure 9: Mesh

We parallelized the loop in Figure 8. The indirection array corresponds to the unstructured

mesh in Figure 9. The mesh has 30269 vertices and 44929 edges. The loop was repeated 500 times. The nodes of the mesh were transformed into a one-dimensional array using Recursive Spectral Bisection-based indexing [19].

The load-balancing algorithm requires an estimate of the current computational resources available on a given processor. There are several ways of estimating the computational resources available to the data-parallel applications on a given processor. One metric we have used is the average computation time per data item. Each processor computes this information by dividing the total time spent on the computation by the number of data elements it owned. This assumes that the variation in computational cost per data unit is relatively small.

Data Size	Workstations					
	1,2,3		1,2,3,4		1,2,3,4,5	
	With MCR	Without MCR	With MCR	Without MCR	With MCR	Without MCR
512	0.0037	0.0042	0.0041	0.0043	0.0045	0.0047
2048	0.0047	0.0052	0.0044	0.0056	0.0054	0.006
16384	0.026	0.031	0.0234	0.0309	0.0229	0.0319
131072	0.2448	0.2594	0.1816	0.2440	0.184	0.2584
1048576	1.8417	1.9646	1.4691	1.9444	1.4294	2.0691

Table 2: Average cost of data remapping (in seconds)

Workstations	1,2	1,2,3	1,2,3,4	1,2,3,4,5
Sort1	0.247	0.171	0.136	0.131
Sort2	0.236	0.169	0.130	0.125
Simple Strategy	0.2	0.188	0.176	0.290

Table 3: Time required for building communication schedule using different strategies (in seconds).

Workstations	Time	Efficiency
1	97.61	1
1,2	55.68	0.88
1,2,3	42.27	0.77
1,2,3,4	34.06	0.72
1,2,3,4,5	31.50	0.62

Table 4: Execution time of the parallel loop for 500 iterations in static environments (in seconds).

Workstations	Execution Time with Load Balancing	Execution Time without Load Balancing	Load Balance Check	Load Balancing Cost
1	290.93			
1,2	88.96	166.2	0.005	0.58
1,2,3	57.22	115.6	0.007	0.39
1,2,3,4	43.52	92.54	0.008	0.19
1,2,3,4,5	40.56	79.32	0.011	0.17

Table 5: Execution time of the parallelize loop for 500 iterations in an adaptive environment (in seconds).

We first measured the performance of the library in a static environment. Table 3 shows the time required to build a communication schedule using the different methods described in Section 3. *Simple Strategy* corresponds to the time for building the communication schedule when an explicit translation table is used (which requires communication). *Sort1, Sort2* correspond to the time for building the communication schedule using Schedule_sort1 and Schedule_sort2, respectively. For a fixed graph, as the number of processors increase, the cost of sorting-based schedules will decrease because the amount of data assigned to each processor decreases. When the number of processors increases, the number of message setups increases, adversely affecting the simple strategy. The time requirements for the latter two schemes can be reduced by improving our current software.

Table 4 gives the execution time of the library in static environments. These results show that a reasonable efficiency can be achieved in most cases.

We used the same environment as above to measure the performance in a controlled adaptive environment. The performance was measured using the following initial conditions:

1. A constant competing load was added to one of the processors (processor 1).

2. The graph was decomposed assuming all the processors had equal computational ratio.

We performed the following experiments:

1. The parallel loop was executed for 500 iterations without any load balancing.

2. The loop was executed for 10 iterations. A check was made after 10 iterations. Using the information gathered for the 10 iterations, a remapping was performed and was used for the remaining 490 iterations.

The results are presented in Table 5. As expected, these results show that using the remapping substantially improves the time required for execution. The cost of load balancing (remapping and building the new communication schedule) is close to the time required for completing a few iterations of the parallel loop, while the cost of performing the load balance check is an order of magnitude lower. These results show that even if a check is done every 10 iterations, the

overhead of performing this check will be small compared to the total execution cost; however, if the environment adapts during that time, the potential advantages of the remapping can be substantial. The frequency of this check and when the remapping should be performed are important parameters for achieving good performance, but are beyond the scope of this paper.

6 Conclusions

In this paper we have presented several optimizations necessary for the parallelization of data-parallel applications on an adaptive and nonuniform computational environment. The library was evaluated on a cluster of workstations using P4 in static and adaptive environments. We showed that our runtime library can be used for effective parallelization in the above environment.

Several methods described in the paper are preliminary approaches for solving the subproblems. We are currently investigating improved methods for achieving similar goals, but at a considerably lower runtime overhead. Although the library was targeted towards solving an unstructured grid on a cluster of workstations, we believe many of the techniques developed in this paper are relevant for efficient solution of other regular as well as irregular data-parallel applications in a nonuniform and adaptive computational environment.

7 Acknowledgements

We would like to thank Joel Saltz for providing the PARTI/CHAOS software and for several insightful discussions.

References

[1] I. Angus, G. Fox, J. Kim, and D. Walker, *Solving Problems on Concurrent Processors*, vol. 2, Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] S. Chang, D. Du, J. Hsieh, M. Lin, R. Tsang, "Parallel Computing Over a Cluster of Workstations Interconnected via a Local ATM Network," technical report, University of Minnesota, 1994.

[3] S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetwork and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, pp. 85–110, May 1990.

[4] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang, "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 462–477, September 1994.

[5] Val Donaldson, Francine Berman, and Ramamohan Paturi, "Program Speedup in a Heterogeneous Computing Network," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 316–322, June 1994.

[6] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox, "Fast Mapping and Remapping Algorithm for Irregular and Adaptive Problems," *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, pp. 279–283, Taipei, Taiwan, December 1993.

[7] Chao-Wei Ou, Manoj Gunwani, Sanjay Ranka, "Architecture-Independent Locality Improving Transformations of Computational Graphs Embedded in K-Dimensions," technical report, Syracuse University, January 1995.

[8] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz, "Software Support for Irregular and Loosely Synchronous Problems," *Proceedings of the Conference on High-Performance Computing for Flight Vehicles*, 1992.

[9] F. Ercal, "Heuristic Approaches to Task Allocation for Parallel Computing," Ph.D. thesis, Ohio State University, 1988.

[10] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, vol. 1, Prentice Hall, Englewood Cliffs, NJ, 1988.

[11] G. Fox and W. Furmanski, "Load Balancing Loosely Synchronous Problems with a Neural Network," 1988.

[12] G. Fox, *Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube,* Ed. M. Schultz, Springer-Verlag, Berlin, 1988.

[13] J. Gilbert, G. Miller, and S. Teng, "A Geometric Approach to Mesh Partitioning: Implementation and Experiments," technical report, Xerox Palo Alto Research Center, 1992.

[14] Bruce Hendrickson and Robert Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," technical report SAND92-1460, Sandia National Laboratories, Albuquerque, NM 87185, 1992.

[15] B. Hendrickson and R. Leland, "An Improved Spectral Load-Balancing Method," *Proceedings of 6th SIAM Conference*, pp. 953–961, 1993.

[16] HPF-2 Scope of Activities and Motivating Applications, "High Performance Fortran Forum," November 13, 1994.

[17] High-Performance Fortran Forum, "High-Performance Fortran Language Specification," January 1993.

[18] M. Kaddoura and S. Ranka, "STANCE: Software Techniques for Adaptive and Nonuniform Computational Environments," technical report, Syracuse University, 1995, in preparation.

[19] Maher Kaddoura, Chao-Wei Ou, and Sanjay Ranka, "Mapping Unstructured Computational Graphs for Adaptive and Nonuniform Computational Environments," to appear in *IPPS: Heterogeneous Computing Workshop*, April 1995.

[20] N. Mansour, "Physical Optimization Algorithms for Mapping Data to Distributed-Memory Multiprocessors," Ph.D. thesis, Syracuse University, NY, 1993.

[21] H. Maini, K. Mehrotra, C. Mohan, and S. Ranka, "Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning," *Proceedings of Supercomputing '94*, November 1994.

[22] Johan De Keyser, Kurt Lust, and Dirk Roose, "Run-Time Load Balancing Support for Parallel Multiblock Euler/Navier-Stokes Code with Adaptive Refinement on Distributed Memory Computers," *Parallel Computing*, vol. 20, pp. 1069–1088, September 1994.

[23] N. Nedeljkovic and M. J. Quinn, "Data-Parallel Programming on a Network of Heterogeneous Workstations," *Proceedings of the First International Symposium on High-Performance Distributed Computing*, pp. 28–36, September 1992.

[24] David M. Nicol and Joel H. Saltz, "Dynamic Remapping of Parallel Computations with Varying Resource Demands," *IEEE Transaction on Computers*, vol. 37, N0. 9, September 1988.

[25] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving Finite Element Equations on Current Computers," *Parallel Computations and Their Impact on Mechanics*, pp. 209–227, 1986.

[26] A. Pothen, H. Simon, and K. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM Journal of Matrix Analysis and Application*, vol. 11, No. 3, 430–352, July 1990.

[27] H. Berryman, J. Saltz, and J. Scroggs, "Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines," *Concurrency: Practice and Experience*, vol. 3, pp. 159–178, June 1991.

[28] B. K. Schmidt and V. S. Sunderam "Empirical analysis of overheads in cluster environments," *Concurrency: Practice and Experience* vol. 6, pp. 1–32, February 1994.

[29] Bruce S. Siegell and Peter Steenkiste, "Automatic Generation of Parallel Programs with Dynamic Load Balancing," *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pp. 166–175, August 1994.

[30] H. Simon, "Partitioning of Unstructured Mesh Problems for Parallel Processing," *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Permagon Press, 1991.

[31] Lawrence Snyder and David G. Socha, "An algorithm producing balanced partitionings of data arrays," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 867–875, April 1990.

[32] R. Williams, "Performance of Dynamic Load-Balancing Algorithm for Unstructured Mesh Calculations," *Concurrency*, vol. 3, pp. 457–481, October 1991.