Techniques to Overlap Computation and Communication in Irregular Iterative Applications

*Antonio Lain and Prithviraj Banerjee**
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Abstract

There are many applications in CFD and structural analysis that can be more accurately modeled using unstructured grids. Parallelization of implicit methods for unstructured grids is a difficult and important problem. This paper deals with coloring techniques to overlap computation and communication during the solution of implicit methods on message passing distributed memory multicomputers. An evaluation of coloring techniques for partitioned unstructured grids is first presented. Results show the importance of using partitioning information during coloring. It is next shown that overlapping computation and communication can be formalized as a generalized coloring problem. Modified coloring algorithms are used for this purpose. The PARTI library [4] has been extended to support non-blocking gather-scatter operations and used in conjunction with these algorithms. Practicality issues are evaluated with experimental results on an Intel Paragon multicomputer.

1 Introduction

There are many important applications in CFD, circuit simulation and structural analysis that can be more accurately modeled using iterations on unstructured grids [20]. Figure 1 shows an example of an unstructured grid. Also, in most cases, implicit methods are preferred to explicit ones in terms of stability and convergence [8, 9]. Implicit methods typically perform the following computation iteratively for a particular ordering of nodes:

$$X_i^{new} = f(X_0^{new}, \ldots, X_{i-1}^{new}, X_i^{old}, \ldots, X_n^{old})$$

as opposed to explicit ones that always use old values. While implicit methods require less time to converge to the final solutions, they still require immense computing times on conventional computers for large problem sizes. Recent advances in message passing distributed memory Massively Parallel Processors (MPP) can deliver the computational power needed for these applications. Nevertheless, the parallelization of an implicit method on unstructured grids, for an MPP system, imposes a significant challenge on the programmer. In particular, the following problems need to be addressed:

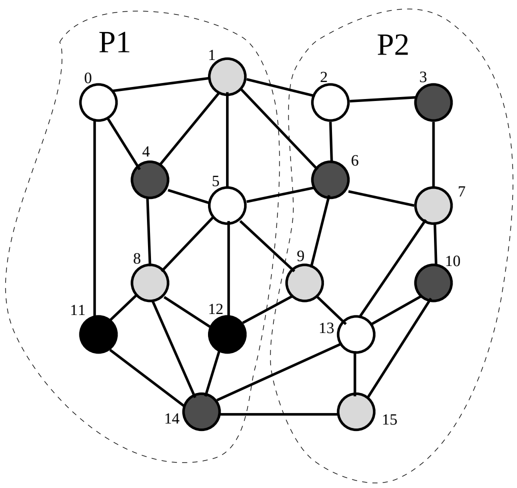

Figure 1: A partitioned and colored unstructured grid

- Partitioning unstructured grids to minimize communication and obtain good load balance.
- Coloring unstructured grids to extract parallelism for implicit methods.
- Providing an efficient run-time support that minimizes communication overhead.

This paper deals with coloring methods to overlap computation and communication in message passing distributed memory multicomputers. Existing coloring techniques and run-time systems are not well suited for this purpose. Therefore, we present extensions to coloring algorithms and a popular run-time system, the PARTI library [4]. Figure 2 illustrates our ultimate goal. Figure 2(a) shows a conventional coloring method where unknowns are reordered into subsets with elements that can be updated in parallel. Figure 2(b) shows an overlapped coloring technique that reorders unknowns into sets such that the computation of elements of a particular color can be overlapped with the communication of elements of the subsequent color. The technique is presented in Section 3.

*This research was supported in part by NASA under contract NASA NAS 1-613, the ONR under contract N00014-91J-1096, and in part by a Fulbright/MEC fellowship.

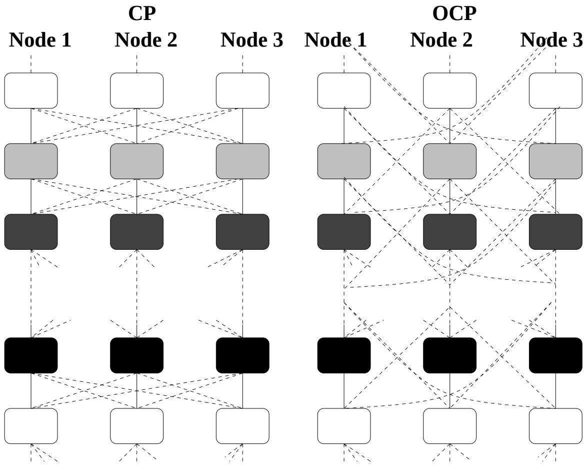

Figure 2: Conventional coloring vs coloring for overlapping

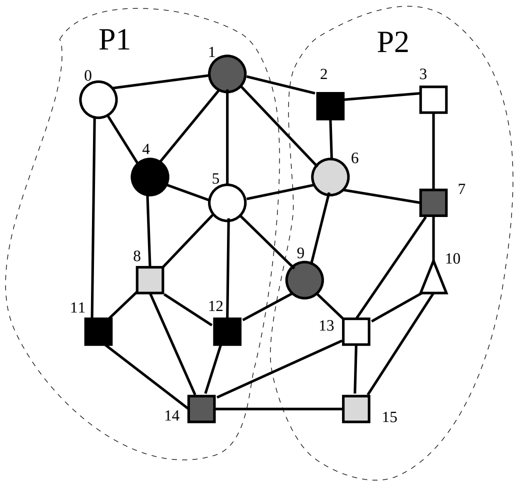

Figure 3: A partitioned and colored for overlapping grid

Many researchers have worked on the partitioning problem [14, 18, 22] and this is currently an area of active research. The main focus of this paper is not partitioning. However, we have used in our experiments a "state of the art" partitioner, a multi-level implementation of Recursive Spectral Bisection [1, 18].

There are several well known coloring techniques to parallelize implicit methods on structured grids [8]. Although the problem of finding the chromatic number of an arbitrary graph is NP-hard [6], reasonably good colorings can be found with simple heuristics [3, 13]. This fact has motivated recent work on extracting parallelism in unstructured problems using these heuristics [16, 25]. A limitation of previous work is that partitioning information is not used during coloring. One contribution of this paper is to evaluate the impact of this limitation on the load balance. Then, we modify the basic algorithms to reduce this imbalance. Figure 1 shows an example of a partitioned and colored unstructured grid partitioned into two groups, P1 and P2. It can be seen that, even though we have the same number of nodes in each partition, the number of nodes for each color across the partitions is not constant.

Run-time support for irregular computations has also been a thoroughly researched topic. One of the most important techniques in this area is Runtime Compilation [17]. The dependency structure of the problem is analyzed, before the actual computation, in a preprocessing step. Preprocessing is needed in irregular problems because the dependency structure is input dependent and cannot be analyzed at compile time. Nevertheless, if a certain structure of the computation is maintained across several iterations, this preprocessing can be reused, amortizing its cost. In practice, this concept is implemented with two sequences of code: an inspector code for the preprocessing and an executor code for the actual computation. This model of computation was supported in the PARTI library [4] and has been incorporated to the ARF [24], KALI [15] and Fortran90D/HPF [2] compilers. Soon it will be part of the PARADIGM [21] compiler at the University of Illinois too.

Multicoloring methods on unstructured grids tend to stress the network more than explicit methods or regular grids. Moreover, it is difficult to obtain perfect load balance for each color across partitions. These problems motivate the second part of this paper: developing coloring techniques that allow the maximum overlap between computation and communication, and relax synchronization requirements across colors. Figure 3 shows our unstructured mesh example colored for overlapping. Provided that the order of execution of colors is fixed, the communication phase of one color can be overlapped with the computation phase of the next one.

It is important to validate the practicality of these coloring techniques. Existing run-time systems for irregular computation implement many important communication optimizations [4, 22]: they aggregate messages with the same destination, eliminate redundant messages, schedule sends and receives to minimize contention in nodes and links [22] and provide low-level, machine dependent communication optimizations [23]. Unfortunately, they do not tend to support non-blocking operations to redistribute data. The approach that we took is to choose a run-time system that supports most of these optimizations, the PARTI library [4], and extend it to support non-blocking communication primitives. Experimental results were obtained on an Intel Paragon multicomputer.

The rest of the paper is organized as follows. Section 2 focuses on the importance of using partitioning information during coloring. Section 3 formalizes the idea of overlapping computation with communication as a generalized coloring problem. Section 4 describes the modifications to the PARTI library that allow non-blocking communication primitives, as well as one application that uses these primitives. Section 5 provides some experimental results that validate the importance of non-blocking primitives. Section 6 contains our conclusions and some discussion of our future research directions.

2 Coloring partitioned graphs

A well-known NP-Hard problem is to find the chromatic number of an arbitrary graph [6]. The definition of chromatic number is the minimum number of colors needed to color the nodes of a graph in such a way, that two nodes connected by an edge have different colors. The relationship between coloring and parallel processing is that nodes that have the same color are independent, therefore, they can be updated in parallel. Figure 4 shows the parallel updating of the colored unstructured grid in Figure 1, assuming that each node is assigned to a different processor. Note in Figure 4 the relationship between the number of colors and the parallelism available in the application.

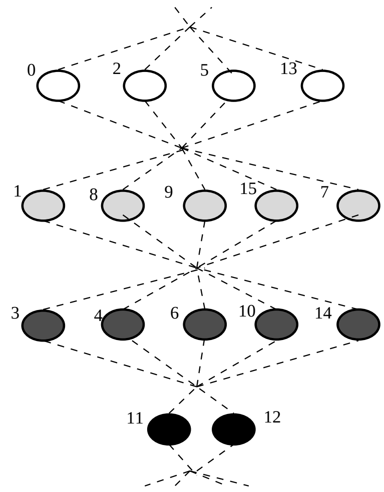

Figure 4: Parallel updating of a colored unstructured grid

An important special case occurs when the graph has already been partitioned. This is the case when partitioning is imposed by previous computations and redistribution is too expensive or we are using an "off-the-shelf" partitioner and we cannot deal with coloring during partitioning. In this case coloring techniques need to use partitioning information to avoid imbalance. Figure 5 illustrates the parallel updating of the partitioned and colored unstructured grid in Figure 1.

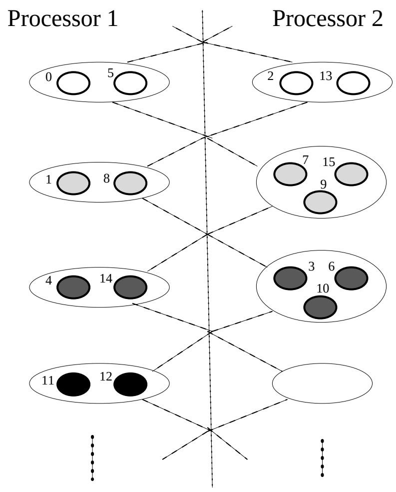

Figure 5: Updating of a partitioned and colored grid

2.1 Quality measures

When we try to color a partitioned graph we have to find a compromise among the following goals:

- Minimize the number of colors to reduce the number of messages and increase their size. Clearly, this minimization also reduces the number of synchronizations.
- Minimize imbalance in the computation of a particular color across partitions. This will also reduce idle time due to synchronization costs.
- If we try to overlap computation and communication of different colors, we also need to minimize computational imbalance across colors.

Our quality measure for the first goal is the number of colors. Quality measures for the other goals are defined next.

Let Cp be the length of the critical path assuming that there is no communication, however, a barrier synchronization is performed after the computation of each color. We assume that P_{ij} is the number of nodes with color i in partition j, and N_c and N_p are the number of colors and partitions, respectively. Cp can be written as:

$$Cp = \sum_{i=1}^{N_c} \max_{j=1}^{N_p} P_{ij}$$

Also, Cp_{opt} is the best Cp that can be obtained with a given partition. Clearly, this is equivalent to have only one color:

$$Cp_{opt} = \max_{j=1}^{N_p} \sum_{i=1}^{N_c} P_{ij}$$

The ratio $Rp = \frac{Cp}{Cp_{opt}}$ represents an architecture and problem size independent quality measure of our second goal.

In the example in Figure 5, $Cp = 2 + 3 + 3 + 2 = 10$, $Cp_{opt} = 8$ and $Rp = \frac{10}{8} = 1.25$.

We can also define Δ_C as the maximum difference of number of nodes between colors:

$$\Delta_C = \max_{i=0}^{N_c} \sum_{k=1}^{N_p} P_{ik} - \min_{i=0}^{N_c} \sum_{k=1}^{N_p} P_{ik}$$

Our third quality measure is $\overline{\Delta}_C$, that is, Δ_C normalized by the average number of nodes per color:

$$\overline{\Delta}_C = \frac{\Delta_C N_c}{\sum_{i=1}^{N_c} \sum_{j=1}^{N_p} P_{ij}}$$

In the example in Figure 5, $\Delta_C = 5 - 2 = 3$ and $\overline{\Delta}_C = \frac{3}{4} = 0.75$

Sometimes the computation cost is not proportional to the number of nodes. For example, in a sweep over the edges of an unstructured mesh, the computation cost is proportional to the number of edges and not to the number of nodes. In these cases, we can associate a different weight to each node and define P^w_{ij} as the sum of the weights of all nodes with color i in partition j. Rp^w and $\overline{\Delta}^w_C$ are defined similarly but using P^w_{ij} instead of P_{ij}. In a sweep over the edges, the weight function is the degree of the node and we get P^e_{ij}, Rp^e and $\overline{\Delta}^e_C$.

2.2 Graph coloring algorithms

In our context, coloring extracts parallelism at run-time during the pre-processing step. Unless this operation is fast, it will be difficult to amortize its cost during the subsequent parallel execution. This fact rules out algorithms that use backtracking techniques or complicated search strategies. Our starting point is a well-known greedy heuristic [3, 25] that has been found to work reasonably well in practice (Basic-Sorted). The idea is to first sort nodes by degree in decreasing order and then try to color them with the first color that produces no collisions. Sorting can be implemented very efficiently using bucket sort because the maximum degree of typical unstructured grids is usually very small compared to the number of nodes.

The Basic-Sorted algorithm does not use either partitioning information or the number of nodes already colored with a particular color. We can extend this algorithm in two ways:

- Choose our initial candidate color depending on the current load.
- Use partitioning information as well as degree information to order the list of nodes.

The first extension to the Basic-Sorted algorithm is to keep track of the number of nodes already assigned to a particular color, and start with the color that has fewer nodes. If that color fails, we just cycle through the rest of the colors starting from that one. We call this extension ColorBalanced-Sorted algorithm (CB-Sorted).

CB-Sorted does not use partitioning information. We can solve this problem if we keep separate utilization records for each partition. When a node is selected for coloring, we find out its partition, and use the utilization statistics for that partition to choose our initial candidate color. A difficulty is that we have to keep consistent the number of colors across partitions. This can be achieved by increasing the number of colors in all the partitions, whenever one partition needs an extra color. This modification to the Basic algorithm will be called ColorPartitionBalanced-Sorted (CPB-Sorted).

CPB-Sorted could still produce load imbalance if all the nodes of one partition are processed before the rest. The reason is that a partition processed later could need extra colors and these color will not have representatives in the first partition. This problem is solved by changing the initial ordering. In particular, we interleave elements of different partitions to ensure that there are uncolored elements of every partition until the end. An orthogonal issue is that nodes can also be sorted by degree in each partition. We call these variants CPB-Interleaved and CPB-Interleaved-Sorted.

2.3 Heuristics evaluation

In this section we evaluate the previous heuristics using partitioned unstructured grids described in Table 1. *Wing* is a three-dimensional finite element grid used for structural analysis of wing-body configuration. *4elt* is a two-dimensional grid representing a four element airfoil [22]. *Rotor* and *Wave* are large three-dimensional grids obtained from RIACS. The rest belong to the Harwell-Boeing collection [5]. We partitioned the grids, in a Sun 4/670, using a trivially modified version of multi-level RSB [1] [1]. The size of the smallest mesh during contraction was set to 500. Table 1 shows that ML-RSB did a reasonable job reducing the number of edges cut ($|E_{cut}|$) and load imbalance across partitions ($\overline{\Delta}_P^e$ [2]).

Table 2 shows our quality measures of different coloring heuristics. Both sorted and unsorted heuristics are evaluated. We can draw the following conclusions from these results:

- The number of colors does not change dramatically, but sorted heuristics are generally better. Basic-Sorted performs best and the number of colors increases with the complexity of the algorithm.
- Sorting by degree in decreasing order improves dramatically the load balance in CB, CPB and CPB-Interleaved. Marginal improvements are also obtained for Basic.
- Basic-Sorted produces imbalanced colorings (Rp^e). Degradation reaches 25 % in some complex grids. Sorted versions of CPB and CPB-Interleaved reduce significantly this overhead. In particular, with CPB-Interleaved-Sorted, this overhead becomes less than 5 % for the largest grids (bcsstk32, rotor and wave). CB-Sorted does not use partitioning information and performs as bad as Basic-Sorted.
- Imbalance across colors ($\overline{\Delta}_C^e$) is very high for Basic. This is expected because trial of colors is always performed in the same sequence. CB-Sorted gives an almost perfect balance in most cases and CPB-Sorted and CPB-Interleaved-Sorted are somewhere in between.
- There is a clear trade-off among these heuristics. Basic-Sorted minimizes the number of colors but creates load imbalance. When we use a more complex heuristic, we get better load balance but more colors. The choice of heuristics will depend on the underlying architecture.

Table 3 shows the time needed to color, in a Sun 4/670, some of the biggest meshes. The code is not tuned for performance and we believe that a factor of 2 of improvement can be achieved. It is important to point out that the time required to partition a grid using ML-RSB (see table 1) is much higher (>20 times) than the time needed to color it. Unless partitioning is done in parallel, little improvement will be obtained from a parallel graph coloring algorithm[13]. The other point that we want to make is that some coloring algorithms that could be thought as more "complex" are sometimes surprisingly faster. The reason is that they tend to succeed in choosing valid colors earlier, amortizing the extra bookkeeping.

Name	Bas	CPB Int	Bas Sort	CB Sort	CPB Sort	CPB I-Sort
bcsstk32	5.2	11.7	11.9	8.07	8.2	14.6
rotor	3.4	11.8	7.3	7.2	7.3	14.6
wave	4.1	17.5	10.4	10.3	10.2	18.6

Table 3: Time for coloring (seconds)

3 Coloring with overlap of communication and computation

By coloring, unknowns are reordered to obtain subsets with elements independent among themselves. The order of execution of these colors is fixed, and this order enforces the new dependencies. We can exploit this ordering to obtain overlapping of computation and communication if we add certain restrictions to coloring.

Given an unstructured mesh represented by an undirected graph (V, E). Assume that nodes colored with col_i, where $i = 0 \dots N_c - 1$, are updated at steps i, $i + N_c$, $i + 2N_c \dots$. We want to find a coloring such that no node

[1] ML-RSB was modified to obtain partitions with equal number of edges instead of equal number of nodes

[2] This is similar to $\overline{\Delta}_C^e$ but using partitioning information

Name	$\|V\|$	$\|E\|$	Avg. Dg.	Max Dg.	$\|E_{cut}\|$	$\overline{\Delta}_P^e$	Time (sec)
bcspwr10	5300	8271	3.121	13	316	0.028	6.4
bcsstk32	44609	985046	44.163	215	60244	0.001	280.2
dwt_2680	2680	11173	8.338	18	1129	0.009	4.9
4elt	15606	45878	5.879	10	1275	0.002	18.0
lshp3466	3466	10215	5.894	6	768	0.010	5.4
rotor	99617	662431	13.299	125	31880	0.0003	132.1
wave	156317	1059331	13.553	44	56586	0.0003	167.3
wing	2851	15093	10.588	17	1386	0.025	5.4

Table 1: Partitioned unstructured grids (16 partitions)

Name	Basic			CPB Interleaved			Basic Sorted			CB Sorted			CPB Sorted			CPB Inter-Sorted		
	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$
bcspwr10	6	1.17	2.02	8	1.22	0.62	5	1.11	1.89	6	1.17	0.00	6	1.00	0.00	6	1.00	0.00
bcsstk32	70	1.30	3.33	70	1.07	0.17	70	1.25	2.88	70	1.22	0.00	70	1.05	0.03	70	1.03	0.02
dwt_2680	8	1.23	1.84	10	1.20	0.98	7	1.23	1.45	8	1.30	0.01	8	1.10	0.03	9	1.12	0.33
4elt	6	1.11	1.65	7	1.05	0.73	6	1.10	1.57	7	1.24	0.69	7	1.17	0.60	7	1.02	0.07
lshp3466	7	1.08	1.70	7	1.06	0.26	7	1.09	1.71	6	1.14	0.00	7	1.11	0.11	7	1.07	0.73
rotor	11	1.13	1.63	14	1.10	0.73	11	1.12	1.62	12	1.15	0.03	12	1.08	0.17	13	1.05	0.10
wave	13	1.09	2.11	15	1.07	0.57	10	1.10	1.66	12	1.25	0.00	12	1.01	0.01	12	1.03	0.02
wing	7	1.10	1.68	11	1.17	0.56	8	1.20	1.81	10	2.03	0.61	9	1.34	0.58	11	1.06	0.03

Table 2: QM of CP Heuristics: Number of colors N_c), imbalance per color/partitions Rp^e and imbalance across colors $\overline{\Delta}_C^e$

with color col_i has an edge to a node colored with col_i, $col_{(i+1)\%N_c}$ or $col_{(i-1)\%N_c}$. Also, this coloring minimizes N_c. This problem is the Overlapping Coloring Problem (OCP). OCP imposes an extra restriction on the Coloring Problem (CP), allowing full overlap of the computation corresponding to the updating of nodes colored with $col_{(i+1)\%N_c}$ with the communication corresponding to the updating of nodes with color col_i with nodes with color $col_{(i+2)\%N_c}$. This overlapping is independent of the partitioning. Figure 2 illustrates the differences between OCP and CP.

Theorem 1 *Given an OCP coloring with N colors, it is possible to obtain a CP coloring with $\lceil \frac{N}{2} \rceil$ colors.*

Proof: Merging col_i with col_{i+1}, where i is even, a new coloring with $\lceil \frac{N}{2} \rceil$ colors is obtained. This coloring is a CP coloring because nodes with color col_i are independent from nodes with color col_{i+1} in the original OCP coloring. □

Unfortunately, the opposite of Theorem 1 is not always true and we have to modify the heuristics presented in Section 2. In particular, we just need to add a more complex test for color collisions.

Table 4 presents results of the OCP coloring heuristics using the grids described in Section 2. Results for Basic and Basic-Sorted are not included because they give trivial solutions. It can be concluded that:

- The number of colors given by these heuristics is close to our theoretical lower bound, twice as many as CP.
- There is some degradation in Rp^e and $\overline{\Delta}_C^e$ with respect to CP. This degradation comes from the added restrictions and the increase in number of colors. The latter cause is more significant in small grids. However, Rp^e becomes less important in OCP because, as a side effect of overlapping, we relax the barrier synchronization.

4 Non-blocking PARTI library

4.1 Motivation

The PARTI library developed by Saltz et al [4] simplifies the implementation of irregular applications on distributed memory multicomputers using the inspector/executor model. PARTI provides support for gathering and scattering globally indexed data elements of irregularly distributed arrays. PARTI is especially efficient when the pattern of communication exchange do not change across iterations. In this case, it is possible to precompute this pattern in a inspector loop, store it in a data structure, and reuse it as many times as needed in the executor loop. Moreover, a variety of communication optimizations are computed in the inspector phase, and they are used later in the executor phase.

One shortcoming of the PARTI library is that gather (scatter) operations are blocking. This means that, in a gather (scatter) operation, a processor has to wait until it has received all its messages. In practice, the majority of the processors are involved in a communication exchange, forcing an implicit barrier synchronization. Blocking forces successful PARTI applications to have:

- Bursty traffic with low average utilization of the network.
- Good load balance between gather operations.

Multicoloring sweeps on unstructured grids do not necessary follow these requirements. In particular, they tend to put more stress on the network than regular grids or explicit methods. Moreover, as it was explained in Section 2, it is difficult to obtain perfect load balance for each color across partitions (Rp). In practice, it is much simpler to obtain good load balance for each sweep across partitions.

4.2 Modifications to the PARTI library

It is possible to relax the previous requirements with simple modifications to the PARTI library. PARTI provides prim-

Name	CPB Interleaved			CB Sorted			CPB Sorted			CPB Inter-Sorted		
	N_c	Rp^e	$\overline{\Delta}^e_C$	N_c	Rp^e	$\overline{\Delta}^e_C$	N_c	Rp^e	$\overline{\Delta}^e_C$	N_c	Rp^e	$\overline{\Delta}^e_C$
bcspwr10	20	1.33	1.48	12	1.32	0.00	13	1.02	0.01	11	1.04	0.01
bcsstk32	150	1.30	1.33	142	1.44	0.00	143	1.12	0.06	141	1.12	0.07
dwt_2680	23	1.32	1.04	18	1.99	0.07	20	1.34	0.64	20	1.35	0.35
4elt	17	1.08	0.85	17	1.34	0.84	16	1.19	0.69	17	1.08	0.75
lshp3466	16	1.18	0.20	15	1.77	0.02	15	1.23	0.53	16	1.15	0.18
rotor	34	1.04	1.09	28	1.27	0.47	28	1.12	0.34	29	1.08	0.20
wave	35	1.07	2.26	29	1.46	0.14	30	1.08	0.42	29	1.14	0.32
wing	24	1.64	0.73	22	2.10	0.88	22	1.93	0.57	24	1.60	1.19

Table 4: QM of OCP Heuristics: Number of colors N_c, imbalance per color/partitions Rp^e and imbalance across colors $\overline{\Delta}^e_C$.

itives to generate schedules (*schedule*), distributed translation tables (*build_translation_table(),dereference()*), to gather or to scatter data of different types with blocking primitives (*gather(data,target,sched),scatter(data,target, sched)*) plus many other advanced features (see [4] for details).

In our extensions, the functionality of gather (scatter) operations is split up in two calls:

*PENDEXCH * gather_nb_begin(data,target,sched)* Implements the first step of a conventional gather but, instead of blocking for incoming messages, it immediately returns a data structure with pending messages identifiers and their buffers.

gather_nb_end(pendExch) Blocks until all the messages described in pendExch arrive. Then, it performs the necessary permutations and bookkeeping and returns.

Internally, few things were needed to change in PARTI to support these split-phase calls:

- We modified the way PARTI uses internal buffers because, with non-blocking operations, it is no longer safe to reuse them.
- FORCE typed messages were not used with non-blocking calls. Otherwise, we need a handshake that defeats the idea of non-blocking.
- New data structures were added to reflect the state of pending communication exchanges.

4.3 An Example

Our example is a multicolored sweep over the edges of an unstructured grid. We are assuming that an OCP algorithm (Section 3) is used for coloring, otherwise data dependences are not respected. Figure 6 shows the skeleton of blocking and non-blocking versions of our example.

In the previous example, the main differences between blocking and non-blocking versions of the code are:

- In the non-blocking code we can have two outstanding gather operations. Obviously, they cannot share the same buffers, so we need two buffers (*odd* and *even*) and alternate their use.
- If the number of colors is odd, we will use different buffers for the same color across iterations. Therefore, we need to compute local indices for both even and odd buffers in the inspector.
- Pre and Post executors are added to handle the special cases of starting and finishing the "communication pipeline".

```
Read_Grid_Description
<Begin Inspector>
  Build_Global_Table
  For each color col_i
    Generate schedule sched_i
    Generate local indices for col_i
    Get number of off-processor
      references (n_i)
  Allocate space for max_i n_i
<End Inspector>
For each iteration
  <Begin Executor>
    For each color col_i
      Gather values with sched_i
      Compute sweep of col_i
  <End Executor>
Write_results
end
```

```
Read_Grid_Description
<Begin Inspector>
  Build_Global_Table
  For each color col_i
    Generate a schedule sched_i
    Generate even local indices for col_i
    Generate odd local indices for col_i
    Get number of off-processor
      references (n_i)
  Allocate space for 2 max_i n_i
<End Inspector>
<Begin Pre-Executor>
  Gather_nb_begin sched_0 in Buf_even
  Gather_nb_begin sched_1 in Buf_odd
  Set State to even
<End Pre-Executor>
For each iteration
  <Begin Executor>
    For each col_i
      Gather_nb_end State
      if State is even
        Compute of col_i with Buf_even
      else
        Compute of col_i with Buf_odd
      Gather_nb_begin sched_(i+2)%NCol
        in Buf_State
      If State is even set State to odd
        Otherwise set State to even
  <End Executor>
<Begin Post-Executor>
  Gather_nb_end even
  Gather_nb_end odd
<End Post-Executor>
Write_results
end
```

Figure 6: Multicolored Sweep: Blocking vs Non-blocking

- The gather operation of a color begins after the computation of another that is two colors ahead ends. This is valid because the coloring technique used (OCP) ensures that nodes with one color are independent from nodes with the previous color.

Figure 7 illustrates the different behaviour of blocking and non-blocking execution on a 16 processor Intel iPSC/2. Tracing information was obtained using PICL [7, 11]. The graphs represent Space-Time communication diagrams of the executor. The vertical axis is the processor number and the horizontal axis time. Horizontal lines represent processor activity (busy or idle) and slanted lines represent messages between processors. As it was expected, the blocking version of the code (left) enforces an implicit barrier synchronization after each color and communication is not overlapped. On the contrary, the non-blocking code has communication patterns that are more irregular in time, and communication gets overlapped with computation.

Figure 8 is a Gantt chart representing activity of processors in time. White means idle time and black means busy time. Although experimental results will be formally presented in the next section, we can see that there is a significant amount of idle time in the blocking code due to synchronization and communication. Non-blocking reduces this idle time using overlapping techniques and looser syn-

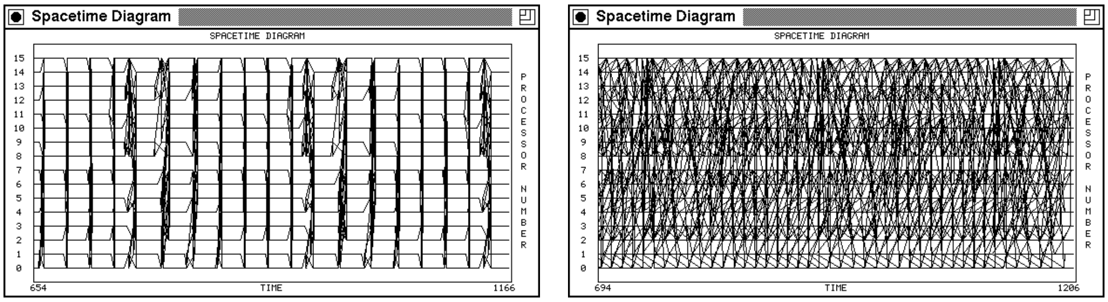

Figure 7: Space-Time Diagram: CP versus OCP on a 16 processor Intel iPSC/2

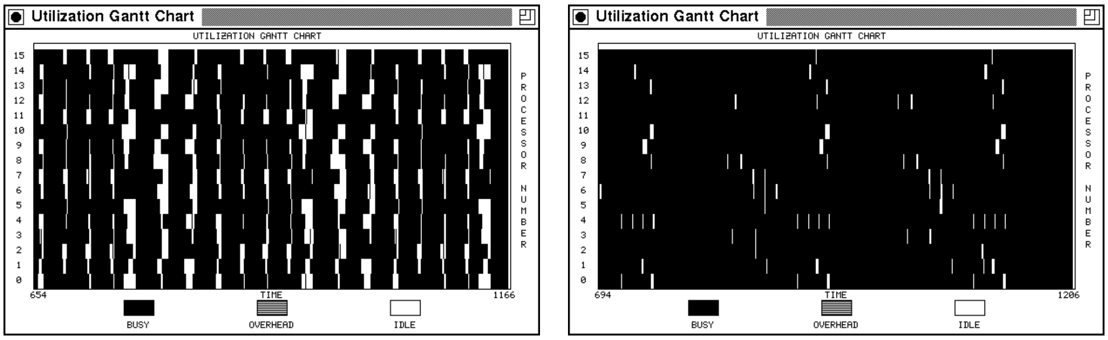

Figure 8: Gantt Chart:CP versus OCP on a 16 processor Intel iPSC/2

chronization.

The effect that we are obtaining is somewhat similar to block prefetching [10] in shared memory architectures. The important difference here is that blocks are obtained from *irregularly* distributed data arrays and cannot be determined at compile time. Therefore, we need to combine the ideas of runtime compilation and block prefetching.

5 Experimental Results

In this section we present two sets of experimental results on a 64 node Intel Paragon multicomputer. The first set shows the intrinsic benefit of using the non-blocking PARTI library. We compare runs that use the same coloring (OCP), but one using the original PARTI library and the other our non-blocking version. Clearly, they both give the same results for each iteration. The second set of experiments targets a comparison between CP and OCP coloring techniques. In this case we are also taking into account the penalty of using a larger number of colors in OCP.

Our application is the one described in Section 4, a multicolored sweep over the edges of an unstructured grid. We did not have access to any real irregular applications operating on those unstructured grids. Even though we accessed the unstructured grids, *wave*, *rotor* and *elt4* from NASA, we were unable to get access to the actual programs corresponding to the grids since they were NASA sensitive applications. We therefore performed two sets of experiments. In the first set the computation in the innermost loop was modeled as an update of each node by a weighted sum of its neighbours. Each edge was assigned a randomly chosen weight. All the operations were in double precision.

In many applications the granularity of the computation in the innermost loop is greater than a simple weighted sum. In our second set of experiments, we performed the update several times to increase granularity. This would model a larger class of irregular applications. We are currently looking at several actual irregular FORTRAN applications which can be mapped on this model [19].

We selected three large grids (*wave*, *rotor* and *elt4*) and a coloring technique that was shown in Section 2 and Section 3 to be reasonably good in most cases (CPB-Interleaved-Sorted). The quality measures of these grids, described in Section 2, for different number of processors are shown in Table 5. We can see from this table that the degradation in load balance due to coloring is very small, although it tends to increase with the number of partitions. Also, OCP has worse load balance than CP due to the increase in number of colors.

Name	Coloring	4	8	16	32	64
4elt	CP	0.69	0.53	0.50	0.48	
	OCP	0.78	0.64	0.76	0.94	
rotor	CP	7.53	4.42	2.70	1.91	1.95
	OCP	7.40	4.39	2.97	3.06	3.66
wave	CP	12.73	6.81	4.07	2.68	2.33
	OCP	12.41	6.99	4.22	3.61	4.16

Table 6: Inspector time (sec)

Although the focus of this paper is not inspector code performance, Tables 6 and 7 compare inspector with executor times for different grids and number of partitions. Partitioning and coloring of the grid are performed sequentially, in a pre-processing stage, and their timings are not included in Table 6. Note that inspector execution is not affected by our modifications to the PARTI library. Therefore, we only need to present results using the original library. We want to point out the following:

- Inspector times are almost independent of the num-

Name	Coloring	4			8			16			32			64		
		N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$	N_c	Rp^e	$\overline{\Delta}_C^e$
4elt	**CP**	7	1.00	0.69	7	1.01	0.16	7	1.02	0.07	7	1.07	0.22			
	OCP	17	1.01	0.68	15	1.03	0.14	17	1.08	0.75	16	1.15	0.65			
rotor	**CP**	12	1.04	0.13	12	1.06	0.14	13	1.05	0.10	12	1.07	0.09	13	1.15	0.20
	OCP	27	1.03	0.17	29	1.05	0.16	29	1.08	0.20	31	1.18	0.58	29	1.24	0.30
wave	**CP**	12	1.00	0.01	13	1.04	0.22	12	1.03	0.02	12	1.05	0.04	12	1.06	0.02
	OCP	30	1.06	0.42	29	1.10	0.28	29	1.14	0.32	29	1.11	0.13	29	1.16	0.32

Table 5: QM of CPB-Interleaved-Sorted: Number of Colors N_c, imbalance per color/partitions Rp^e and imbalance across colors $\overline{\Delta}_C^e$

Name	Coloring	4	8	16	32	64
4elt	**CP**	2.26	1.41	1.03	0.81	
	OCP	2.55	1.72	1.36	1.15	
rotor	**CP**	31.86	16.88	9.30	6.11	4.86
	OCP	32.09	17.81	10.98	8.58	6.85
wave	**CP**	48.34	25.46	13.76	8.58	6.60
	OCP	52.31	28.24	16.58	11.5	12.2

Table 7: Blocking PARTI. Executor Time (sec):20 iterations and 10 ops per edge

ber of colors (CP versus OCP) when a small number of processors is used. When the number of processors increases, execution times become nearly proportional to the number of colors. This is explained by the decrease of the ratio of computation to communication, during the execution of the inspector, when the number of processors increases.

- In most cases, a few iterations of the executor are needed to amortize the inspector overhead. This is encouraging because the number of iterations will be much larger in typical iterative applications.

In the rest of this section we will ignore inspector execution and we will use only executor times to compute speed-up values. Moreover, nodes in the Paragon have 32 Mbytes of physical memory and *rotor* and *wave* do not fit into a single processor physical memory. In order to avoid superlinear speed-up due to paging, we obtain an approximation of the sequential execution time ($T_1 = T_p + T_{seq}$) using the Amdahl's law:

$$T_n = \frac{T_p}{n} + T_{seq}$$

and for a 4 and 8 node configuration we get the system of equations:

$$\begin{cases} T_4 = \frac{T_p}{4} + T_{seq} \\ T_8 = \frac{T_p}{8} + T_{seq} \end{cases}$$

Solving this system we obtain an approximation of T_1:

$$T_1 = T_p + T_{seq} = 7T_4 - 6T_8$$

The approximation of T_1 for the original PARTI library with conventional coloring is used to obtain all the estimated speed-up figures of *wave* and *rotor*. Real speed-up curves are obtained for *elt4*, a smaller grid.

Figure 9 shows the results of our first set of experiments comparing the original PARTI library with a blocking sweep application with our modified library and non-blocking sweep. The same OCP coloring is used in both cases, giving identical results for each iteration. The following conclusions can be drawn:

- Non-blocking exchange primitives improve performance significantly. This is is also true when the granularity of computation is increased. As it was commented in Section 4, this benefit comes from relaxing barrier synchronization and overlapping computation and communication.
- *elt4* shows poor speed-up. The reason is that *elt4* is almost an order of magnitude smaller than the other grids and its granularity is too fine. Clearly, speed-up increases when the granularity is increased.

Experiments with increased granularities were also run (50 and 100 operations per edge) obtaining better overall speed-up, as expected, and showing the same trend as in Figure 9. Results are omitted due to limited space.

Figure 10 presents results for our second set of experiments. In this case we wanted to compare a CP coloring using a blocking library/application with an OCP coloring with a non-blocking library/application. It should be noted that the speedups for both CP and OCP coloring are with respect to the sequential implementation of the CP coloring on a single processor (estimated for *wave* and *rotor*). According to Table 5, OCP coloring more than doubles the number of colors of CP coloring. This translates into many more messages with smaller size. In architectures like the Intel Paragon the communication start-up time, mainly due to software overhead, is large. Therefore, for small messages, communication cost is almost proportional to this start-up time and the number of messages. As it can be seen in Figure 10, for a small number of processors, this increase in communication cost dominates, and Blocking-CP does better than Non-Blocking-OCP. When we increase the number of processors, load imbalance and synchronization costs increase and the flexibility of the non-blocking library/application pays off. The crossover point when the Non-Blocking-OCP is better than Blocking-CP is between 16 and 32 processors in most cases.

In this set of experiments the improvement due to non-blocking is not as significant as in the previous case. We believe that with a larger machine configuration or a lower overhead message passing library the benefit would become more clear. Also, the highly tuned CP coloring heuristic is eliminating most of the imbalance for these grids (see Table 2) limiting the scope of improvements.

Convergence rates of iterative algorithms can be affected by coloring. In this paper we are not dealing with convergence but an encouraging result in [12] shows that convergence tends to improve when the number of colors increases. This fact could also make OCP coloring better than CP coloring in terms of convergence. Nevertheless, further experimentation is needed to validate this conjecture.

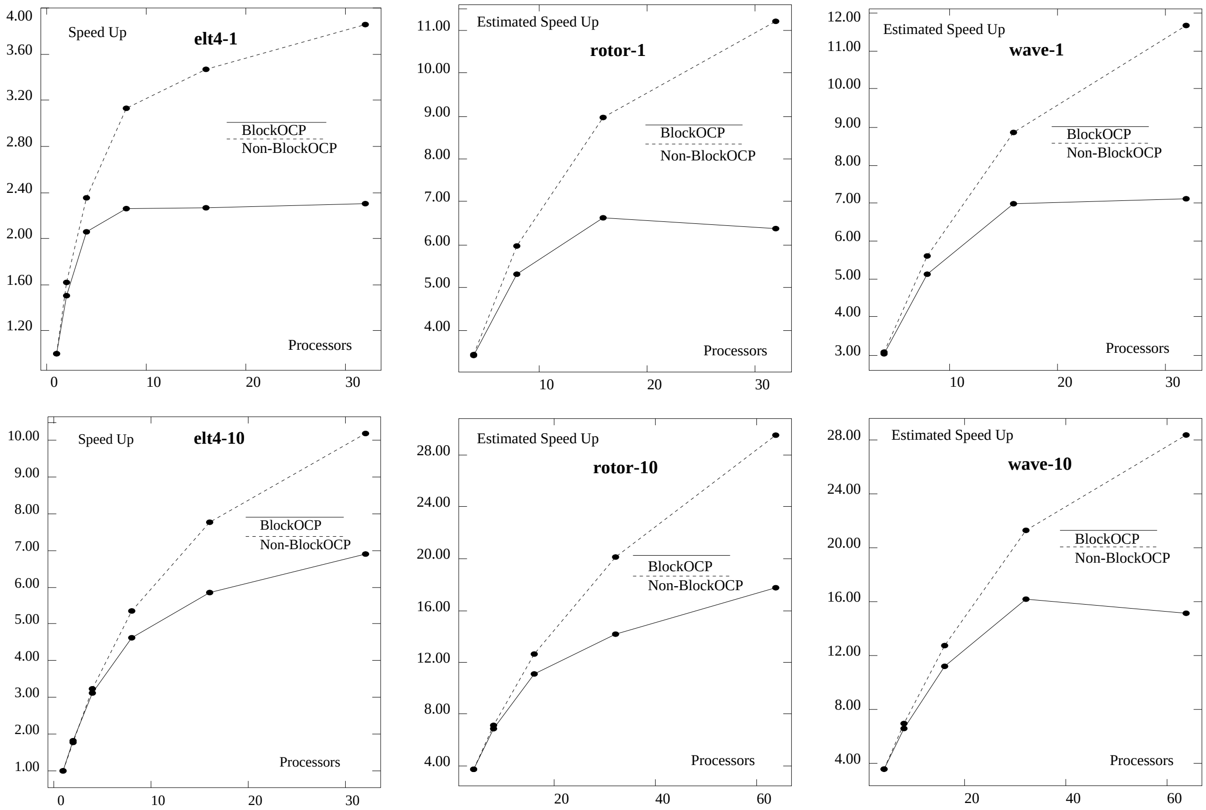

Figure 9: Executor Speed-up: Non-blocking improvement (1 and 10 op per edge)

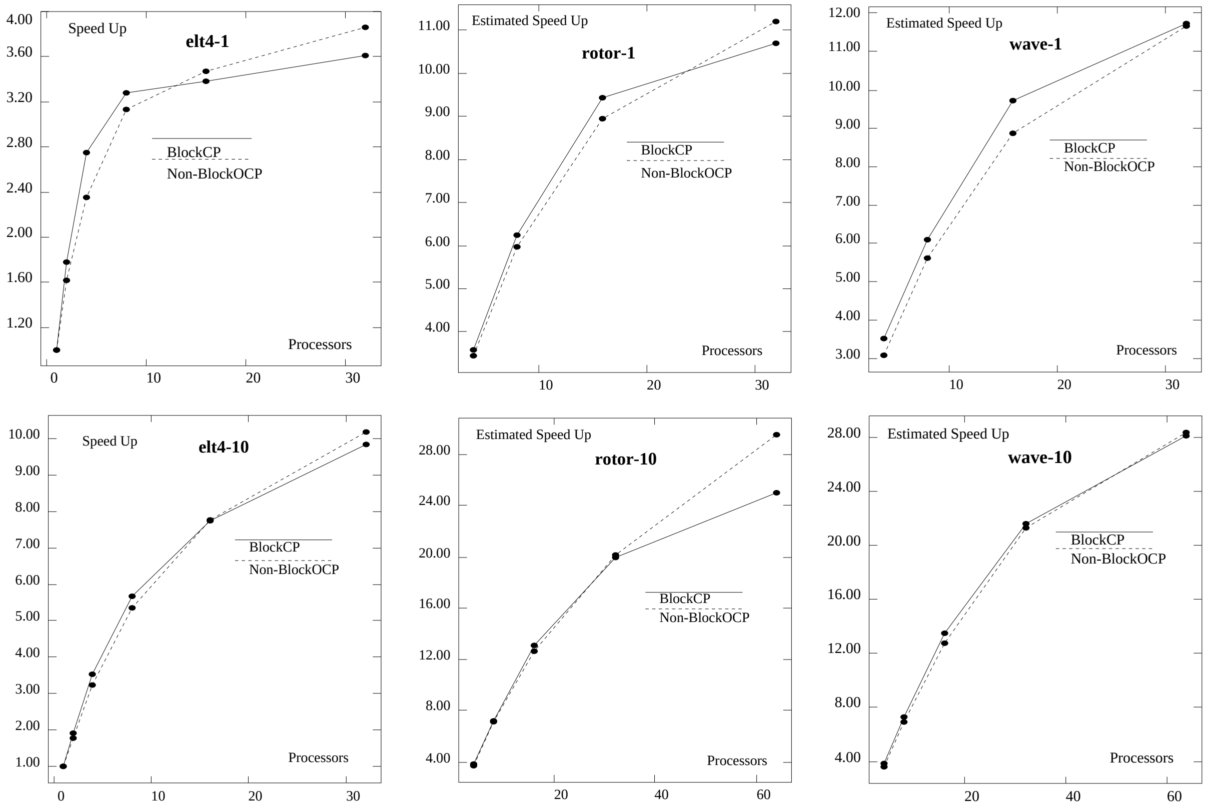

Figure 10: Executor Speed-up: CP versus OCP (1 and 10 op per edge)

6 Conclusions

This paper deals with coloring methods to overlap computation and communication during the solution of implicit methods on message passing distributed memory multicomputers. We have presented and evaluated coloring heuristics that take into account partitioning information. It has been shown that these coloring techniques reduce load imbalance across colors and load imbalance for a particular color across partitions. Moreover, the number of colors required does not increase dramatically.

Also, it has been shown that the overlapping of communication and computation can be formalized as a generalized coloring problem. Heuristics for this problem are obtained with simple modifications of coloring heuristics. It is validated that these modified heuristics work reasonably well in practice.

We show the practicality of these overlapping techniques extending the PARTI library to allow non-blocking communication primitives and developing an application on top of it. This application uses the generalized coloring ordering to implement a sweep over the edges of an unstructured mesh.

Results on a 64-node Intel Paragon clearly show the benefit of non-blocking communication exchange primitives. Also, for a large enough number of processors, overlapping coloring techniques are better than conventional ones, although the increase in number of colors, and therefore the number of messages, can reduce their applicability to smaller configurations.

Our future plans are to automate some of these techniques in the PARADIGM compiler at the University of Illinois [21]. In particular, we want to automatically generate inspectors in HPF programs and insert non-blocking communication primitives whenever this is possible.

7 Acknowledgments

The authors would like to thank Horst D. Simon and Steve Barnard for providing us with their ML-RSB code and Joel Saltz for making available to us the PARTI library code. R. Schreiber at RIACS made available to us through anonymous ftp some of the test grids. We are grateful to the San Diego Supercomputing Center for letting us use the Intel Paragon.

References

[1] Stephen T. Barnard and Horst Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames Research Center, November 1992.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

[3] N. Christofides. An algorithm for the chromatic number of a graph. *The Computer Journal*, (14):38–39, 1971.

[4] Raja Das and Joel Saltz. *A Manual for PARTI Runtime Primitives*. IACS University of Maryland, December 1992.

[5] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Users's guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, October 1992.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[7] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[8] Gene Golub and James M. Ortega. *Scientific Computing: an Introduction with Parallel Computing*. Academic Press, 1993.

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2nd edition, 1989.

[10] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. Technical Report 996, CSRD, May 1990.

[11] Michael T. Heath and Jennifer E. Finger. *ParaGraph: a Tool for Visualizing Performance of Parallel Programs*. Oak Ridge National Laboratory.

[12] Mark T. Jones and Paul E. Plassmann. The effect of many-color orderings on the convergence of iterative methods. Technical Report MCS-P292-0292, Argonne National Laboratory, May 1992.

[13] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal of Scientific Computing*, 14(3):654–669, May 1993.

[14] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, February 1970.

[15] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[16] Rami G. Melhem and K.V.S. Ramarao. Multicolor reordering of sparse matrices resulting from irregular grids. *ACM Transactions on Mathematical Software*, 14(2), June 1988.

[17] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. Technical Report ICASE 90-59, Institute for Computer Applications in Science and Engineering, Hampton, VA, September 1990.

[18] H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[19] Horst Simon, 1993. Private Communication.

[20] Horst D. Simon, editor. *Parallel Computational Fluid Dynamics*. The MIT Press, 1992.

[21] E. Su, D. J. Palermo, and P. Banerjee. Automating parallelization of regular computations for distributed memory multicomputers in the PARADIGM compiler. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II,30–38, August 1993.

[22] V. Venkatakrishnan and Horst D. Simon. A MIMD implementation of a parallel Euler solver. *Journal of Supercomputing*, 6(2):117–127, 1992.

[23] T. von Eicken, D. E. Culler, and S. C. Goldstein. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[24] Janet Wu, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. Technical Report 91-13, ICASE, January 91.

[25] Stavros A. Zenios and John M. Mulvey. A distributed algorithm for convex network optimization problems. *Parallel Computing*, pages 45–56, 1988.