Partitioning an Array onto a Mesh of Processors

Fredrik Manne and Tor Sørevik

Department of Informatics, University of Bergen,
N-5020 Bergen, Norway
email: {fredrikm,tors}@ii.uib.no

Abstract. Achieving an even load balance with a low communication overhead is a fundamental task in parallel computing. In this paper we consider the problem of partitioning an array into a number of blocks such that the maximum amount of work in any block is as low as possible. We review different proposed schemes for this problem and the complexity of their communication pattern. We present new approximation algorithms for computing a well balanced generalized block distribution as well as an algorithm for computing an optimal semi-generalized block distribution. The various algorithms are tested and compared on a number of different matrices.

1 Introduction

A basic task in parallel computing is the partitioning and subsequent distribution of data to processors. The problem one faces in this operation is how to balance two often contradictory aims; finding an equal distribution of the computational work and at the same time minimizing the imposed communication.

In the data parallel model this can be modeled as a graph partitioning problem where the vertices represents data and the edges indicate that results obtained from processing one data unit will be needed for further processing of the other. Finding an optimal solution is know to be NP-hard [8], and hence impossible to solve to optimum for large instances.

In settings where locality is of importance the partitioning and resulting mapping should as far as possible be done such that adjacent nodes are mapped to the same processor. Thus the dataset should be partitioned into connected components. If the data is stored in an array these components might for reasons of both efficiency and simplicity be restricted to be rectangular blocks of the array. Several high-performance computing languages include the possibility for the user to specify such a partitioning and distribution of data onto a logical set of processors. The compiler then maps the data onto the physical processors and determines the communication pattern. An example of one such scheme is the block distribution found in languages such as Vienna Fortran [4] and HPF [10].

In general the block distribution will result in equal size blocks and therefore cannot adapt to load imbalance that might be present. Consider ocean modeling where the presence of land gives irregular areas for which no computations are needed. As demonstrated in [2] a block distribution that takes this into account reduces the time spent on a parallel computation. More general partitioning schemes that have been proposed for these kinds of problems include the generalized and semi-generalized block distribution [5, 15, 16, 17].

In this paper we discuss a number of different partitioning schemes. In particular, we describe an efficient iterative algorithm that computes a well balanced generalized block distribution. We also show how an optimal semi-generalized block distribution can be found. The performance of these algorithms are compared with other orderings such as the uniform block distribution and the binary recursive decomposition [1, 3]. The algorithms presented extend earlier work for one dimensional arrays [13, 14].

The paper is organized as follows: Section 2 gives formal definitions of the different partitioning schemes and relate these to each other. Section 3 presents new algorithms for computing the different distributions and Section 4 reports on the performance of these. Finally, in Section 5 we conclude and point to areas of further work.

2 Structured Distributions

In this section we define and relate the different types of distributions and discuss what kind of communication pattern they impose. One measurement of the communication complexity is the maximum number of neighbors a block can have. In this paper we only consider arrays of dimension two. All results, however, may easily be extended to arrays of higher dimensions.

Let $A \in \Re^{m \times n}$ and let p and q be integers such that $1 \leq p \leq m$ and $1 \leq q \leq n$. Let $R = \{r_o, r_1, ..., r_p\}$ be integers such that $1 = r_0 \leq r_1 \leq ... \leq r_p = m + 1$. Then R defines a *partitioning* of $[1..m]$ into p consecutive intervals $[r_i, ..., r_{i+1} - 1]$, $0 \leq i < p$. We denote this interval by $[r_i, ..., r_{i+1}]$. A partitioning of $[1..m]$ into p intervals and of $[1..n]$ into q intervals defines a partitioning of A into $p \times q$ blocks.

We now define the different types of distributions of A. The distributions are given in increasing order of complexity.

2.1 Non Recursive Distributions

The most simple distribution we consider is the uniform distribution:

Definition 1 Uniform Block Distribution. The interval $[1..m]$ is partitioned into p consecutive intervals of size $\lceil \frac{m}{p} \rceil$ with the possible exception of the last interval. Similarly $[1..n]$ is partitioned into q intervals of size $\lceil \frac{n}{q} \rceil$.

The uniform distribution divides A into $p \times q$ equally sized blocks. See Figure 1a for an example. In certain applications the amount of data that needs to be communicated is proportional to the perimeter of each block. In this setting the uniform distribution minimizes the time needed for communication. If the work associated with each element of A is equal it also gives a perfectly balanced workload. But being fixed a priori it has no possibility to adapt to load imbalance if the computational work varies throughout A. This might be mitigated by moving any of the $p + q - 2$ interior delimiters. By doing so we allow more flexibility in the size of the blocks while at the same time keeping the regular communication pattern of the uniform block distribution.

Definition 2 Generalized Block Distribution (GBD). The interval $[1..m]$ is partitioned into p consecutive intervals without restrictions on the size of each interval. Similarly $[1..n]$ is partitioned into q intervals.

See Figure 1b for an example of the GBD. The GBD was discussed by Fox et. al. [7] and implemented as part of Superb environment [17] and later in Vienna Fortran [6]. It is also a candidate to be included as part of the ongoing HPF2 effort [11]. For examples of how the GBD can be used in areas such as sparse-matrix and particle-in-cell computations see [5] and [12].

While the GBD has the same structured communication pattern as the uniform distribution the blocks sizes vary. The time spent on communication is therefore likely to be higher than with the uniform distribution.

In some cases it is only necessary to have a structured distribution in one dimension. If this is the horizontal direction we may relax the partitioning conditions in the vertical direction. One would thus allow for an individual partitioning of the columns in each row segment given by the horizontal distribution.

Definition 3 Semi-Generalized Block Distribution (SBD). The interval $[1..m]$ is partitioned into p consecutive intervals $[r_i, r_{i+1}]$, $1 \leq i \leq p$ without restrictions on the size of $r_{i+1} - r_i$. For each horizontal interval $[r_i, r_{i+1}]$ the interval $[1..n]$ is partitioned into q intervals.

See Figure 1c for an example of the SBD. Ujaldon et. al. proposed a partitioning scheme called Multiple recursive decomposition which results in a SBD [15, 16]. It was designed for solving problems from sparse linear algebra on parallel computers.

The SBD has a possibility to adapt better to load imbalance than the GBD. But since a block may have as many as $2q + 2$ neighbors the communication pattern becomes less structured.

In a parallel environment the time spent on a computation is determined by the processor taking the longest time. To estimate the time needed to process each block we define a non-negative cost function ϕ on contiguous blocks of A. The assumptions on ϕ are that if a and b are blocks of A such that $a \subseteq b$ then $\phi_a \leq \phi_b$, and $\phi_a = 0$ if and only if a is the empty block. For most reasonable functions ϕ we expect that if the value of a (or b) is known then the value of b (or a) can be computed in $O(|b| - |a|)$ time. An example of ϕ might be the number of non-zero elements or the sum of the absolute values of the elements in a block.

We now get the following optimization problem:

Partition A in such a way that $\max_{i=1:p, j=1:q} \phi_{i,j}$ is minimized.

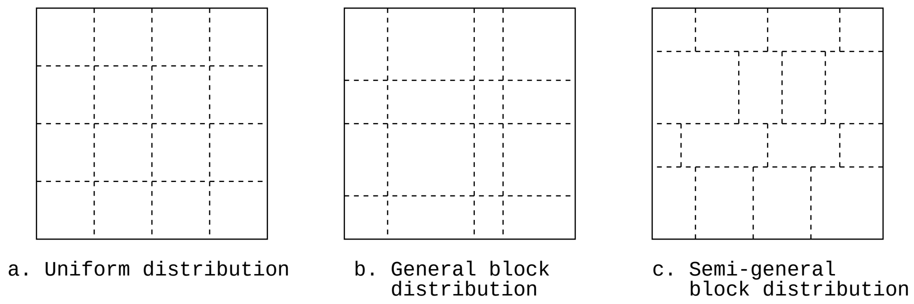

Fig. 1. Examples of the different distributions with $p = q = 4$

The relationship between the optimal values for each of the different distributions is captured in the following:

Theorem 4. *Let A, p, q, and ϕ be defined as above. Let further $\mathcal{S}$ be the set of all SBDs on A, $\mathcal{R}$ the set of all GBDs, and $\mathcal{U}$ the uniform distribution. Then the following is true:*

$$\min_{\forall \mathcal{S}} \max_{i=1:p,j=1:q} \phi_{i,j} \leq \min_{\forall \mathcal{R}} \max_{i=1:p,j=1:q} \phi_{i,j} \leq \max_{(i,j)\in\mathcal{U}} \phi_{i,j} \tag{1}$$

Proof: The result follows trivially from the fact that $\mathcal{U} \in \mathcal{R} \subseteq \mathcal{S}$. □

As is evident from Theorem 4 the more unstructured the distribution is the more even we can get the load balance, but as discussed above, the more complex and time consuming the communication becomes.

2.2 Recursive Distributions

It is possible to generalize Definitions 2 and 3 to make the distributions recursive. The partitioning would then recursively be applied to each block for a number of d levels. Thus A is partitioned into $p^d \times q^d$ blocks. The relationships given by Theorem 4 still holds true for the recursive distributions. Note also that for both the GBD and the SBD the minimum cost of the recursive distribution is lower than the minimum cost of the non-recursive version partitioned into $p^d \times q^d$ blocks.

The maximum number of neighbors any block can have above or below and to the right or left is q^{d-1} and p^{d-1} for the recursive GBD and q^d and p^{d-1} for the recursive SBD. Thus the recursive orderings give more complicated communication patterns than their non-recursive counterparts.

For $p = q = 2$ the recursive SBD gives the well known binary recursive decomposition [1]. Figure 2 shows examples of the recursive GBD and the binary recursive decomposition.

Less restricted block distributions than the ones presented here may lead to a better load balance but are likely to give more irregular communication patterns that would be difficult to implement efficiently.

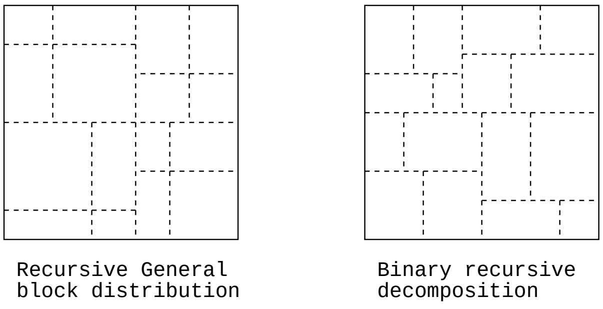

Fig. 2. Recursive distributions with $p = q = d = 2$

3 Algorithms

In this section we describe an efficient iterative algorithm for computing a well balanced GBD. The solution obtained is shown to be a local optimum. We also show how an optimal SBD can be found. For completeness we describe a number of proposed approximation algorithms.

All presented algorithms extend previous results on partitioning a one dimensional array. We therefore start by recapturing this problem and its solution.

The problem is identical to the generalized partitioning problem discussed so far but with $n = q = 1$. Thus we are partitioning a vector of length m into p consecutive intervals. The current fastest algorithm for solving this problem is based on dynamic programming and runs in time $O(p(m-p))$ [14]. This is based on the same assumptions on ϕ as for the general problem. Thus the cost function can be computed in time $O(1)$ when the size of an interval changes by one. Every function evaluation in the algorithm is of this type. Thus if the time to calculate the cost function is c the time complexity becomes $O(p(m-p)c)$.

3.1 The Generalized Block Distribution

Consider first the problem of partitioning $[1..n]$ into q intervals when the partitioning of $[1..m]$ into p intervals has been fixed. The placement of vertical delimiters i and $i+1$ then defines p blocks of cost $\phi_{i,j}$, $1 \leq j \leq p$. We define a new cost function $\theta_i = \max_{1 \leq j \leq p} \phi_{i,j}$. The function θ has the same monotone properties as ϕ. Thus we can reduce the placement of the vertical delimiters to solving the one-dimensional case with cost function θ. Using the dynamic programming algorithm we can now find an optimal placement of the vertical delimiters. This is also true if the vertical delimiters are fixed and an optimal placement of the horizontal ones is desired.

We suggest an algorithm where this step is applied iteratively: The delimiters are fixed in one direction and placed optimally in the other. This is then repeated while alternating which delimiters are fixed until no decrease in the maximum cost is obtained. At this stage a local optimum has been reached, where moving

0	4	10	3
6	12	3	24
0	15	20	30
15	24	20	6

Fig. 3. A local optimum which is not globally optimal.

any set of either horizontally or vertically delimiters will not decrease the overall cost of the partition.

However, as the example in Figure 3 shows, the solution obtained might not be globally optimal. Here $p = q = 2$ and the cost function is the sum of the elements. The indicated solution of cost 76 cannot be improved by moving any one of the delimiters, whereas the optimal solution has the lower right hand block of size one and is of cost 70.

With the assumptions made on ϕ in Section 2 the time complexity of calculating θ when the size of an interval changes by one is $O(m)$. Thus the time complexity of one iteration of algorithm becomes $O(qm(n-q) + pn(m-p))$.

For most natural cost functions it is possible to improve the time complexity by collapsing parts of the array in each iteration. The assumption we make is that the contribution to the cost function from one column (or row) of a block can be reduced to one number. Given a partition of the rows we can then collapse each column segment to one number. This reduces the size of the matrix from $m \times n$ to $p \times n$ and the time complexity becomes $O(mn + pq(n-q) + pq(m-p))$ where the mn term comes from the collapsing step.

We note that if the contribution from a single row or column segment $[i, j]$ can be calculated as $\phi(1..j) - \phi(1..i)$ then the rows can be collapsed in $O(pn)$ time and the columns in $O(qm)$. This is done by pre-computing $\phi(1..i)$ for every value of i for every row and column. The expense for this speedup is that we need an $O(mn)$ pre-computational step and extra storage to hold the values from this step.

A more simple approximation algorithm for computing a GBD was presented in [12]. In this algorithm the rows and columns are collapsed and partitioned separately giving a time complexity of $O(p(m-p) + q(n-q) + mn)$.

3.2 The Semi-Generalized Block Distribution

In this section we show that the dynamic programming algorithm can be extended to compute an optimal SBD.

Consider the row interval $[r_i, r_{i+1}]$ in a SBD. Let γ be the value of an optimal q-partition of the column segments in this interval. Since ϕ is monotone it follows that γ is also monotone and $\phi = 0$ if and only if $\gamma = 0$. Thus we can use the dynamic programming algorithm to compute an optimal p-partition of $[1..m]$ using γ as cost function resulting in an optimal SBD.

The time complexity of finding the optimal q-partition of $[r_i, r_{i+1}]$ is $O(q(n-q)(r_{i+1} - r_i))$. The function γ needs to be evaluated $O(p(m-p))$ times and therefore the overall time complexity becomes $O(pqm(m-p)(n-q))$.

This result can be improved if it is possible to collapse the columns of A. Recall that in the dynamic programming algorithms the function value of an interval is always obtained after one of the delimiters of the interval has been moved exactly one place. Thus if we performed a collapsing of the columns the previous time we evaluated γ we can update this in time $O(n)$ to the value needed in the current evaluation. This reduces the time complexity of evaluating γ to $O(n + q(n-q))$ and the overall time complexity becomes $O(p(m-p)(n + q(n-q))) = O(pq(m-p)(n-q))$.

An approximation algorithm for computing a SBD can be obtained by first determining an optimal partition on the collapsed rows of A. The columns of each row segment are then collapsed and optimally q partitioned. The time complexity of this algorithm is $O(mn + p(m-p) + pq(n-q))$.

For completeness we also describe the Multiple recursive decomposition [16]. Let $p_1, p_2, ..., p_k$ be the prime factorization of p in descending order of magnitude. The rows are first partitioned into p_1 intervals such as to minimize the cost of the most expensive interval. Each interval is then further partitioned into p_2 intervals and so on until p intervals have been obtained. This process is then repeated for each row segment using the prime factors of q. We note that if p and q are powers of 2 and it is possible to collapse the columns and rows of A, the time complexity of this algorithm is $O(m \log p + pn \log q + mn)$.

4 Numerical experiments

We have implemented the algorithms of Section 3, and performed a number of experiments in order to investigate how well they partition an array with respect to load balance. The cost function we have used is the sum of the elements. For each data set we report the results from the following algorithms:

- **SBD** The optimal SBD distribution as described in Section 3.2.
- **ASD** The approximation algorithm described in Section 3.2.
- **MRD** The Multiple recursive decomposition as described in Section 3.2.
- **GBD** The iterative algorithm from Section 3.1 for computing a GBD.
- **GBA** The simple approximation algorithm mentioned in Section 3.1.
- **UD** The uniform distribution.
- **BRD** The binary recursive decomposition [1].

To illustrate the behavior of the algorithms we tried them on 3 different types of test matrices. The function $rand()$ generates numbers from a uniform random sequence of non-negative integers less than $2^{15} - 1$.

1. **Skewed matrix:** The weights of the elements are skewed to the bottom right of the array. The matrix elements are: $a_{ij} = (rand() \bmod 7)(i + j)$
2. **Peak matrix:** The matrix has a distinct peak, randomly chosen at (r, c). The matrix elements are: $a_{ij} = (rand() \bmod 127)/((\mid r - i \mid)(\mid c - j \mid) + 1.0)$
3. **Diagonal dominant matrix:** The matrix elements are: $a_{ij} = (rand() \bmod 127)/((\mid i - j \mid) + 2.0)$

For each type of matrix we have performed three series of experiments. First we keep the size of the matrix and the value of q fixed, while p is increased. We then increase both p and q while keeping the size of matrix fixed and finally we keep both p and q fixed while changing the size of the matrix. The results from the tests are presented in Table 1 through 3. For all test matrices the results are normalized relative to $(\sum_{i,j} a_{i,j})/(pq)$ which is the theoretical lower bound for any distribution.

The algorithm for computing the GBD distribution has been initialized in three different ways: (i) Starting with the delimiters as far left as possible, (ii) as far right as possible, and (iii) the Uniform distribution. The results obtained differ only marginally from each other. We therefore only report the best result for each test case. The number of iterations needed before a converged solution is obtained varies from 3 to 25 with an average of 6. The number of iterations is the sum of the number of vertical and horizontal partitionings performed. There does not appear to be any correlation between the number of iterations and the goodness of the obtained solution.

Problem size				Semi-general dist.			General dist.			
m	n	p	q	SBD	ASD	MRD	GBD	GBA	UD	BRD
256	256	2	2	1.01	1.01	1.01	1.06	1.06	1.50	1.00
256	256	4	2	1.01	1.01	1.01	1.06	1.11	1.64	
256	256	8	2	1.02	1.02	1.02	1.07	1.17	1.66	
256	256	16	2	1.04	1.04	1.06	1.09	1.24	1.72	
256	256	32	2	1.06	1.07	1.12	1.13	1.20	1.78	
256	256	64	2	1.12	1.12	1.25	1.18	1.28	1.75	
256	256	4	4	1.01	1.02	1.01	1.09	1.15	1.73	1.02
256	256	8	8	1.04	1.04	1.05	1.11	1.23	1.88	1.04
256	256	16	16	1.06	1.07	1.08	1.18	1.32	1.93	1.10
256	256	32	32	1.16	1.17	1.23	1.32	1.55	2.15	1.23
256	256	64	64	1.30	1.30	1.66	1.65	1.92	2.50	1.50
256	32	16	16	1.27	1.35	1.41	1.35	1.44	2.40	1.42
256	64	16	16	1.16	1.19	1.24	1.27	1.42	2.19	1.26
256	128	16	16	1.10	1.11	1.13	1.20	1.39	1.95	1.12
256	256	16	16	1.07	1.07	1.12	1.18	1.32	1.93	1.10

Table 1. Results from test using skewed matrices.

As expected the SBDs give a better load balance than the GBD.

Problem size				Semi-general dist.			General dist.			
m	n	p	q	SBD	ASD	MRD	GBD	GBA	UD	BRD
256	256	2	2	1.01	1.01	1.02	1.01	1.04	1.50	1.01
256	256	4	2	1.02	1.01	1.03	1.11	1.14	2.25	
256	256	8	2	1.02	1.07	1.02	1.18	1.33	3.16	
256	256	16	2	1.04	1.07	1.07	1.24	1.57	4.50	
256	256	32	2	1.10	1.10	1.27	1.31	1.37	6.44	
256	256	64	2	1.19	1.21	1.53	1.26	1.37	4.87	
256	256	4	4	1.02	1.06	1.02	1.23	1.40	4.41	1.04
256	256	8	8	1.04	1.07	1.05	1.35	2.27	6.94	1.06
256	256	16	16	1.12	1.21	1.34	1.65	3.15	16.00	1.48
256	256	32	32	1.39	1.58	1.70	1.75	3.33	23.43	2.19
256	256	64	64	9.93	9.93	9.93	9.93	9.93	25.16	9.93
256	32	16	16	1.40	1.63	1.96	1.68	2.11	9.32	1.63
256	64	16	16	1.27	1.54	1.54	1.67	2.16	8.73	1.41
256	128	16	16	1.16	1.26	1.33	1.48	3.34	9.27	1.32
256	256	16	16	1.12	1.27	1.55	1.55	2.73	14.05	1.24

Table 2. Results from tests using peak matrices.

Problem size				Semi-general dist.			General dist.			
m	n	p	q	SBD	ASD	MRD	GBD	GBA	UD	BRD
256	256	2	2	1.00	1.01	1.00	1.66	1.67	1.69	1.00
256	256	4	2	1.01	1.01	1.02	1.68	1.76	1.68	
256	256	8	2	1.03	1.04	1.03	1.68	1.80	1.79	
256	256	16	2	1.04	1.04	1.06	1.68	1.80	1.79	
256	256	32	2	1.09	1.09	1.11	1.75	1.87	1.86	
256	256	64	2	1.12	1.17	1.18	1.81	1.98	1.92	
256	256	4	4	1.02	1.02	1.02	2.02	2.91	2.74	1.03
256	256	8	8	1.06	1.06	1.09	3.22	4.98	4.31	1.07
256	256	16	16	1.14	1.17	1.26	5.02	8.15	6.47	1.20
256	256	32	32	1.50	1.70	1.87	7.63	13.07	9.40	1.61
256	256	64	64	3.93	4.72	4.94	11.08	18.71	14.34	3.02
256	32	16	16	1.81	2.59	2.59	2.68	5.33	9.17	1.68
256	64	16	16	1.37	1.51	1.86	3.51	5.47	7.74	1.65
256	128	16	16	1.21	1.22	1.44	4.08	6.69	6.71	1.41
256	256	16	16	1.13	1.13	1.30	5.26	8.03	6.58	1.25

Table 3. Results from tests using diagonal dominant matrices.

For almost every test problem the optimal SBD is fairly close to the lower bound given by the average cost. The most noticeable exception is the peak matrix of size 256×256 with $p = q = 64$. In this case the most expensive block consists of one single matrix element for the optimal SBD as well as for the iterative GBD. Thus the load imbalance we see here is inherit in the problem.

The ASD distribution is never far from the lower bound given by the optimal SBD. Compared with the lower time-complexity of the ASD this might make it a good choice. It also gives better load balance than the MRD distribution. While the binary recursive decomposition places in between ASD and MRD.

The GBD distribution outperforms the GBA distribution. This must, however, be compared with the higher time complexity of computing the GBD distribution.

For both the skewed and the peak matrices the GBD distribution is fairly close to the optimal SBD. From Theorem 4 it follows that for these matrices the presented GBD must be close to optimal. For the diagonally dominant matrices the difference is larger. However, we believe this to be a feature inherit in the definition of the GBD and not a consequence of our algorithm.

In general we see that it becomes harder to obtain a well balanced distribution as the ratios $\frac{m}{p}$ and $\frac{n}{q}$ become smaller.

It follows from the test results for both the SBD and the GBD that the more time one is willing to spend on obtaining a good distribution the better the load balance becomes.

As expected the results confirm that the uniform distribution is not suitable for matrices with non-uniform load.

5 Conclusion

We have presented an efficient iterative algorithm that computes a well balanced GBD. We have also developed an algorithm that computes an optimal SBD. These were tried on a number of test problems and compared with other approximation algorithms. This showed that the SBDs in general gave a more even load balance than the GBDs. This must, however, be compared with the more complicated communication pattern given by the SBD.

When choosing a distribution one must first determine what kind of communication needs one has. Based on this and the criticality of achieving an even load balance one can decide which type of distribution to use. Then depending on how much time one is willing to spend on calculating a distribution one can decide which algorithm to use.

An advantage of the GBD is that it is easy to specify, only requiring two vectors of length q and p whereas the SBD requires $p + p * q$ data elements.

In a resent development it has been shown that computing the optimal GBD is NP-hard for certain cost functions [9].

As a continuation of this work we are currently implementing several sparse matrix algorithms on a parallel computer. The object is to investigate how well the different partitioning schemes behave on real-world problems where both load balance and communication influence the overall time.

References

1. M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Comput., C-36 (1987), pp. 570–580.

2. R. Bleck, S. Dean, M. O'Keefe, and A. Sawdey, *A comparison of data-parallel and message-passing versions of the Miami Isopycnic Coordinate Ocean Model (MICOM)*, Parallel Comput., 21 (1995), pp. 1695–1720.
3. S. H. Bokhari, T. W. Crockett, and D. M. Nicol, *Parametric binary dissection*, Tech. Rep. ICASE Report No. 93-39, Nasa Langley Research Center, 1993.
4. B. Chapman, P. Mehrotra, and H. Zima, *Programming in Vienna Fortran*, Sci. Prog., 1 (1992), pp. 31–50.
5. ———, *High performance Fortran languages: Advanced applications and their implementation*, Future Generation Computer Systems, (1995), pp. 401–407.
6. ———, *Extending HPF for advanced data parallel applications*, IEEE Trans. Par. Dist. Syst., (Fall 1994), pp. 59–70.
7. 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.
8. M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.
9. M. Grigni and F. Manne, *On the complexity of the generalized block distribution.* To appear in the proceedings of 1996 Workshop on Irregular Problems, 1996.
10. High Performance Fortran Forum, *High performance language specification. Version 1.0*, Sci. Prog., 1–2 (1993), pp. 1–170.
11. *High Performance Fortran Forum Home Page.* http://www.crpc.rice.edu/HPFF/home.html.
12. F. Manne, *Load Balancing in Parallel Sparse Matrix Computations*, PhD thesis, University of Bergen, Norway, 1993.
13. F. Manne and T. Sørevik, *Optimal partitioning of sequences*, J. Alg., 19 (1995), pp. 235–249.
14. B. Olstad and F. Manne, *Efficient partitioning of sequences*, IEEE Trans. Comput., 44 (1995), pp. 1322–1326.
15. M. Ujaldon, S. D. Sharma, J. Saltz, and E. Zapata, *Run-time techniques for parallelizing sparse matrix problems*, in Proceedings of 1995 Workshop on Irregular Problems, 1995.
16. M. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zima, *Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation.* Submitted to IEEE Trans. Par. Dist. Syst.
17. H. Zima, H. Bast, and M. Gerndt, *Superb: A tool for semi-automatic MIMD/SIMD parallelization*, Parallel Comput., (1986), pp. 1–18.

This article was processed using the LaTeX macro package with LLNCS style