An Object-Oriented Recursive Bisection Algorithm for CM5

Gregor von Laszewski
gregor@npac.syr.edu

April 15, 1993

Contents

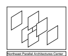
1 Introduction **1**
- 1.1 Object Oriented Programming . . . 2
- 1.2 Message Passing Model . . . 3
  - 1.2.1 The CM5 . . . 3

2 Recursive Bisection Algorithm **4**
- 2.1 Sorting . . . 5
  - 2.1.1 Sequential Quicksort Algorithm . . . 6
  - 2.1.2 Parallel Mergesort Algorithm . . . 6
- 2.2 Parallel Recursive Bisection . . . 8

3 Results **9**

4 Conclusion **10**

A Sample Code **12**



An Object-Oriented Recursive Bisection Algorithm for CM5

Gregor von Laszewski

gregor@npac.syr.edu

Abstract

Recursive bisection is a very fast partitioning strategy. It is often used for dividing points in a two dimensional plane into a number of partitions containing an approximately equal number of points. The membership of a point to a specific partition is specified by its location in the two dimensional plane. This scheme can be generalized in two ways.

First, the bisection algorithm can be expanded to multidimensional planes.

Second, the algorithm can be used on any plane for which a linear order between the elements is defined.

This report presents an object-oriented approach using a multi dimensional recursive bisection algorithm on a distributed memory architecture. By specifying the number of planes and their linear order for each plane a variety of real problems can be solved with minimum effort.

Results for partitioning two dimensional planes on the CM5 are given.

Keywords: Bisection, Parallel Sorting, Object Oriented Programming.

1 Introduction

Recursive bisection is a very fast partitioning strategy. It is often used for dividing points in a two dimensional plane into a number of partitions containing an approximately equal number of points. The membership of a point to a specific partition is specified by its location in the two dimensional plane. This scheme can be generalized in two ways.
First, the bisection algorithm can be expanded to multidimensional planes.
Second, the algorithm can be used on any plane for which a linear order between the elements is defined.
This report presents an object-oriented approach using a multi dimensional recursive bisection algorithm on a distributed memory architecture. By specifying the number of planes and their linear order for each plane a variety of real problems can be solved with minimum effort.
One realistic example application is described in [3] and shows the usability of the here described method in a real application. In some VLSI circuit simulations the computational core is dominated by the simulation of the transistors. To increase the speed of the simulation the goal is to distribute the calculations onto differen processors of a parallel machine. To achieve load balancing the

recursive bisection technique is used. On each processor should be mapped an equal number of transistors for the simulation to achieve a minimal execution time for the simulation.
The report is structured in the following way:
First, the motivation for an object oriented programming approach is given. Than, the message passing model necessary for the implementation is described, followed by the specification of a parallel sorting algorithm [6] which builds the core of the recursive bisection algorithm described in the next section. In the last section results calculated on the CM5 are presented.
The Figure 1 tries to outline the hierarchical structure of the program. It should be noted that a user of the object oriented recursive bisection algorithm is only responsible for defining the objects and their linear order in the different dimensions in order to adapt it his specific problem.

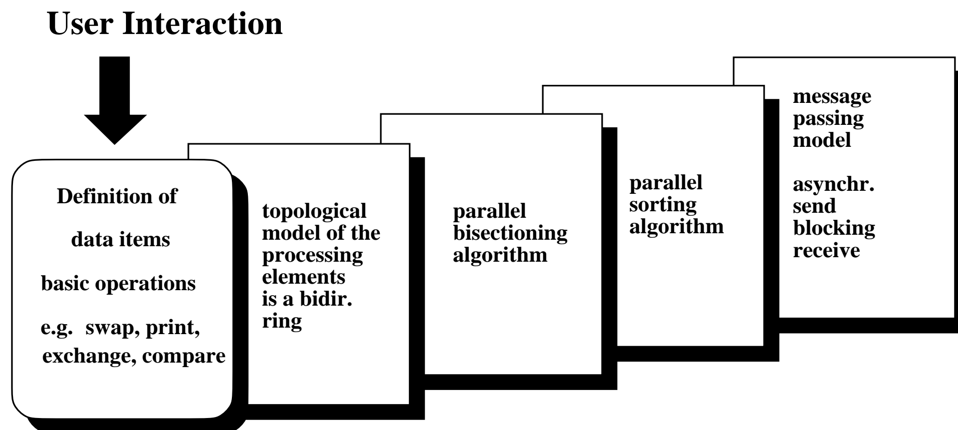


Figure 1: The Logical division of the bisection package

1.1 Object Oriented Programming

Since the invention of the computer the way programs are written changed drastically. Overcoming the use of programming in binary machine codes with structured programming languages like *C* and *PASCAL* the productivity and quality of writing programs by software engineers increased a lot.
Nevertheless, with the increasing complexity of a programming task even those programming languages are difficult to use for large software engineering projects. Object oriented programming languages seem to be a good solution to overcome this obstacle. Problems are decomposed into subgroups including both code and data. By specifying different data objects the same code can be executed on these data without changing the code itself.
Instead of choosing existing object oriented programming languages this study still uses the programming language *C* because part of the code is expected to be included into other code written in this language.

1.2 Message Passing Model

Beside the flexibility of the algorithm provided by the object oriented programming approach the algorithm is portable on many machines. This is guaranteed since the algorithm uses only a few general intercommunication routines. These routines can be ported on different machine platforms, for example the CM5, the iPSC860 [5], the nCUBE2 or even a workstation of networks.
In general the processing elements should be ordered in a bidirectional ring as shown in Figure 2. This is possible on most of the existing MIMD machines since they provide routines enabling arbitrary sending and receiving messages between two specified computation nodes.

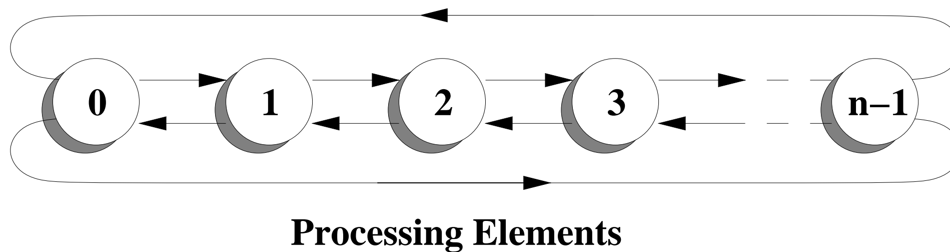


Figure 2: Topological order of the processing elements

The *send* command should block the calling process until the message is send out safely. The send routine should allow to specify an arbitrary destination from the set of available. The routine used in the program code has the following syntax:

SendMsg (message, length in bytes, message type, destination processor)

The semantic of the routine is straight forward. A message starting at the address of *message* of *length in bytes* is send to the *destination processor*. The message is distinguished by its *message type*.
The corresponding *receive* routine receives a message of specified type and length from the calling process until the message is received. The calling process is blocked till the message arrived and is stored at the specified memory location.
The routine used in the program code has the following syntax:

RecvMsg (message, length in bytes, message type, source processor)

1.2.1 The CM5

The CM5 can operate in both SIMD and MIMD mode. For the implementation shown here only the MIMD mode is used. The available machine has 32 nodes each with 32 megabyte of memory. Each node includes a RISC processor as well as four vector units capable of 128 MFLOP peak performance.

The RISC processors are 33MHz SPARC processors. They use 64 KByte cache for instructions and data together. The Processor is rated with 22 Mips and 5 MFLOPS.
The interconnection network between processing elements is given by a fattree shown in Figure 3. The CMMD message passing library provides the above mentioned sending and receiving commands.

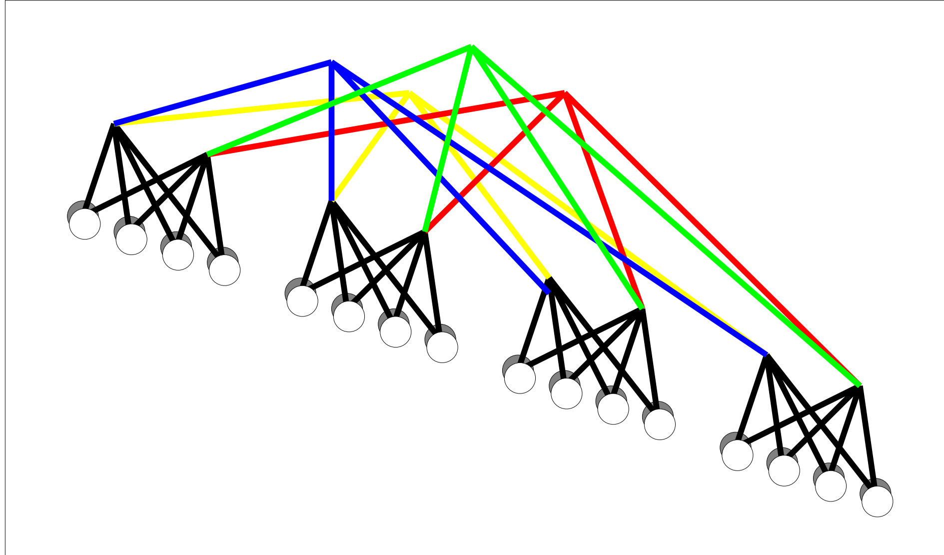


Figure 3: The interconnection network of the CM5

For the recursive bisection algorithm a bidirectional ring is mapped onto the topology in such a way that processors with consecutive processing ID's are neighbors to each other. Therefore, only these neighbors are exchanging messages over the interconnection network.
For the implementation on the CM5 the CMMD message passing routines are used.

2 Recursive Bisection Algorithm

As shown in Figure 4 a sorting algorithm is the building block of the parallel bisection algorithm. After introducing the sequential and parallel sorting algorithm the parallel bisection algorithm is described in detail in a later section.
To explain the recursive bisection algorithm the following example illustrates the algorithmic technique. Assume in a two dimensional finite plane a number of random points are located. The problem is to map on a given number of processors an equal number of points. Figure 4. shows a two dimensional plane with 100 points.
First, the algorithm places half of the points to the left and the other half to the right. This can be done by simply sorting the elements in the x-dimension. In the next step the same is done in each of the parts separately in the y-dimension. This can be repeated as long as the desired number of

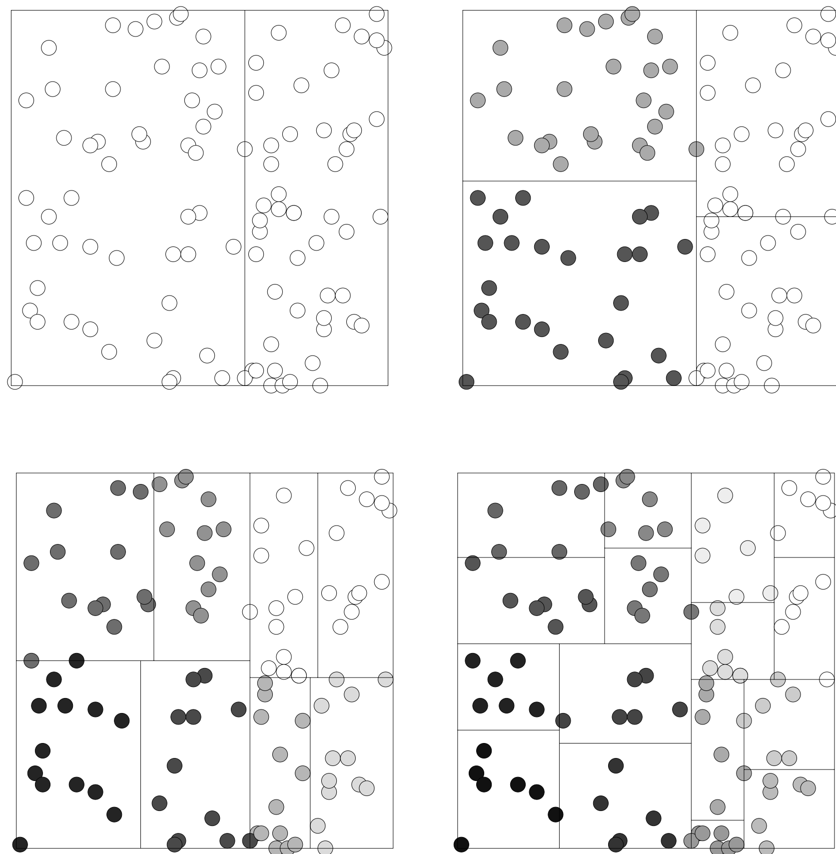


Figure 4: Partitioning a random graph with 100 points graph into 2, 4, 8, and 16 parts

partitions are found. In the Figure results for 2, 4, 8, and 16 partitions are given.

2.1 Sorting

A fast sorting algorithm is the basis of the recursive bisection strategy. The sorting algorithm is dependent on the topology assumed on the processing elements. To be most general only a bidirectional array is assumed as topology for the sorting process. It is important that the sorting process includes only part of the computational units so that the other processors might work independently on another task. This enables one to use a parallel machine for example for more than one independent sorting processes as used in the parallel bisection algorithm introduced later. The parallel sorting algorithm itself uses the sequential quicksort in its initial step. Therefore, the quicksort algorithm is described first, followed by the description of the parallel merge sort algorithm.

2.1.1 Sequential Quicksort Algorithm

Quicksort is one of the most popular sequential sorting algorithms due to its good average time complexity of $O(n \log n)$ for large problem sizes. The quicksort algorithm is based on the important fact that exchanges are executed over large distances which leads to high efficiency.

First, the median of a sequence of items is found. This partitions the initial sequence into two subsequences. The subsequences have the property that they are either larger or smaller than the median. This step is repeated recursively on the so generated subsequences till no further divide-and-conquer is possible. A pseudo-code fragment representing the quicksort algorithm is shown in Figure 5.

For a detailed description and complexity analysis one might consult [4].

```
PROCEDURE quicksort (int l, int r, array a)
   i = l;
   j = r;
   x = a_{(l+r)/2}
   REPEAT
      WHILE (COMPARE (a_i,x) = SMALLER) i=i+1;
      WHILE (COMPARE (x,a_j) = SMALLER) j=j-1;
      IF (i≤j) then
         SwapItem (a_i, a_j);
         i=i+1;
         j=i-1;
      ENDIF
   UNTIL (i>j);
   IF (l<j) then  quicksort(l,j,a);
   IF (i<r) then  quicksort(i,r,a);
END PROCEDURE
```

The routine *COMPARE* returns SMALLER if the value of the first argument is smaller than the second argument.

Figure 5: The sequential quicksort algorithm

2.1.2 Parallel Mergesort Algorithm

The parallel sorting algorithm is based on the technique described in [1, 2] under *Merge-Splitting Sort*. This algorithm is based on the Odd-Even Transposition Sort but takes into account that not every element can be stored on a separate processor. This is a very realistic restriction since MIMD machines are build only with a medium number of processing elements. An alternative approach would be to use data parallel programming languages. In this report we concentrate on the message passing model.

The example shown in Figure 6 illustrates the algorithm.

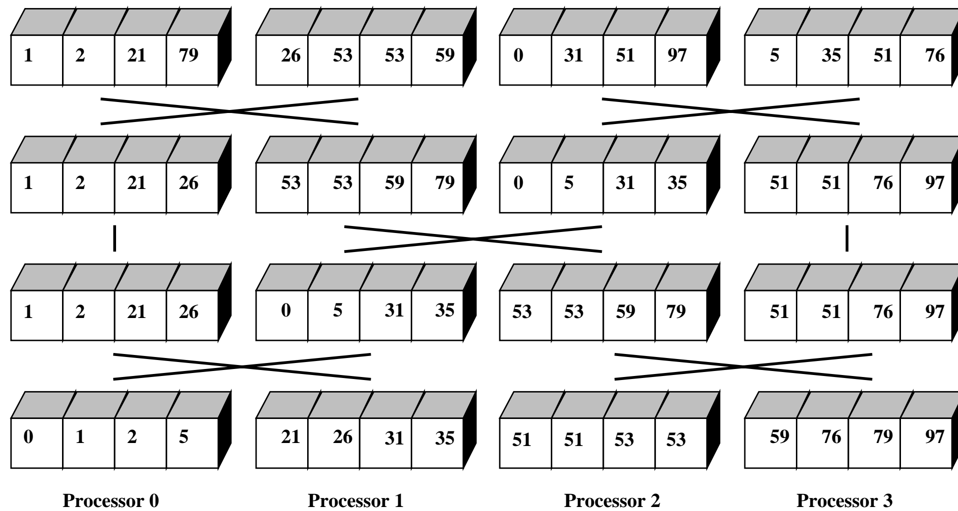


Figure 6: Example of the parallel sorting algorithm on 4 processors and 16 data items

On the four processors 16 data items are generated in random order. These data items are sorted with the sequential sorting algorithm. The result is shown in the first row of Figure 6.
The next step, includes the sending of all element of processors with even numbers to the right and with odd numbers to the left. Each processing element consist now of two data sequences. Its own and the one from its corresponding neighbor. On each even processor the smaller elements of the two data sequences are determined and for the odd ones the larger ones. They replace the data items stored in the processor. This is shown in the Figure 6 in the second row.
The following step is similar to the above one with the difference that now each odd processor sends its data to the left and the even ones to the right. This is done with two exemption. The processors on both ends of the linear array do not participate in the exchange process. Than, on each odd processors the smaller elements and on each even the bigger ones are collected.
These two steps are repeated $p/2$ times in order to guaranty the correctness of the sorting algorithm, where p specifies the number of processors.

Complexity Analysis The sorting of n numbers with the sequential quicksort algorithm in each processor is done in $O(\frac{n}{p}\log\frac{n}{p})$ steps. Transferring the data elements from one processor is done in $O(\frac{n}{p})$ steps. A mergesort of two lists requires at most $2\frac{n}{p}$ steps. Thus, the two steps are done in $O(\frac{n}{p})$ steps. Since they are repeated $p/2$ times, the total running time is

$$t(n,p) = O\left(\frac{n}{p}\log\frac{n}{p}\right) + O(n)$$

In contrast to [1, 2] the following mergesort algorithms are used in the hope to improve the average

running time for the mergesort algorithm with the factor 2. The worst case complexity analysis maintains unchanged.

The procedure *mergehigh* mergesorts two lists from the highest to the lowest element till the new list has half the elements from both of the lists.

The procedure *mergelow* mergesorts two lists from the lowest to the highest till the new list has half the elements from both of the lists.

The Figure 7 shows the parallel mergesort algorithm in pseudo code.

```
SendRecv (Destination)
  send the own items to destination and receive the neighbor items

PROCEDURE mergesort-parallel
    FOR (t=0; t< ProcessorsInArray; t++)
        IF (even(t)) THEN
            IF      (even(me)) THEN        SendRecv(right);  mergelow(dimension);
            ELSE IF (odd(me)) THEN         SendRecv(left);   mergehigh(dimension);
            ENDIF
        ELSE
            IF ((me = FirstProcessor) or (me = LastProcessor)) THEN
                no msg exchange
            ELSE
                IF      (even(me)) THEN    SendRecv(left);   mergehigh(dimension);
                ELSE IF (odd(me)) THEN     SendRecv(right);  mergelow(dimension);
                ENDIF
            ENDIF
        ENDIF
    END FOR
END PROCEDURE
```

Figure 7: The parallel mergesort algorithm

2.2 Parallel Recursive Bisection

With the help of the above described procedures the Recursive bisection algorithm is given in Figure 8 with the functionality as described at the beginning of Section 2 .

The bisection algorithm is initiated with the call:

bisection $(0, p-1, 0, \text{level}, \frac{n}{p})$;

The variable level determines that the input data is partitioned into 2^{level} parts.

```
PROCEDURE bisection (FirstProcessor, LastProcessor,
                     dimension, level, ProcItems)
    quicksort (0, ProcItems-1, items, dimension);
    midpoint = ( LastProcessor - FirstProcessor ) / 2;
    mergesort-parallel ( FirstProcessor, LastProcessor, dimension);
    IF level > 1 THEN
        NextDimension = (dimension + 1) % MaxDimension;
        bisection (FirstProcessor, midpoint,   NextDimension,level-1,ProcItems);
        bisection (midpoint+1,   LastProcessor,NextDimension,level-1,ProcItems);
    END IF
END PROCEDURE
```

Here the quicksort algorithm is extended to the different dimensions of the input data.

Figure 8: The parallel mergesort algorithm

3 Results

The experiments are conducted on a CM-5 with 32 nodes. The bisection algorithm has been tested for various data structures. To compare results with the ones found in literature here timings on bisecting random points in a two dimensional plane are given.

The number of points are varied between approximately 64 and 61000. The problems are solved using different numbers of processors. The Figures 9 and 10 show the result using 2,4,8,16 and 32 processors for partitioning the input data into 2 and 4 partitions.

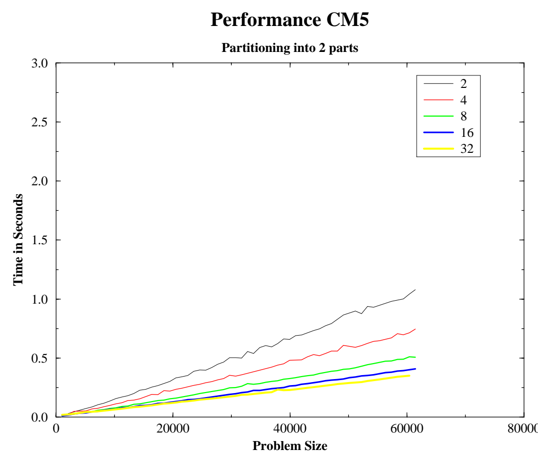


Figure 9: Performance of the bisection algorithm on the CM5 with vectorunits. A random graph is partitioned into 2 parts.

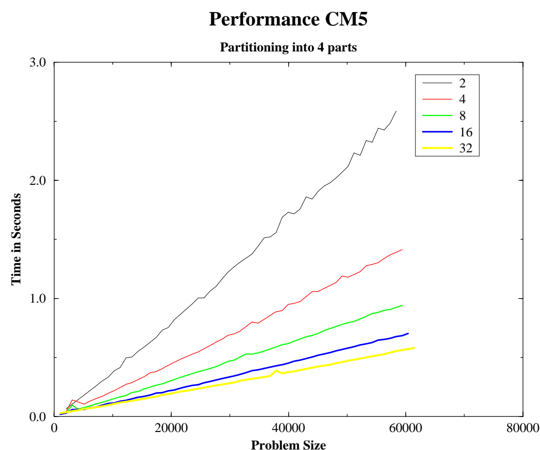


Figure 10: Performance of the bisection algorithm on the CM5 with vectorunits. A random graph is partitioned into 4 parts.

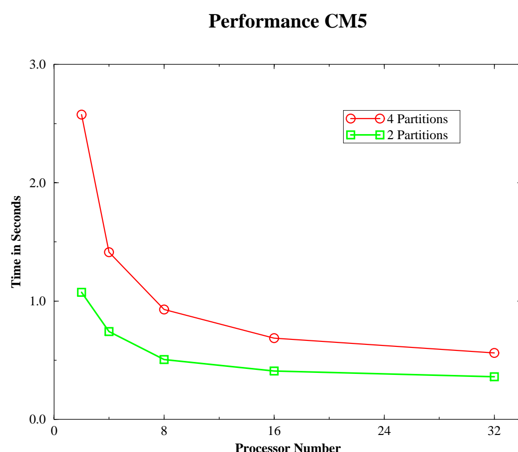


Figure 11: Performance of the bisection algorithm on the CM5 with vectorunits. A random graph with 61440 nodes is partitioned into 2 and 4 parts.

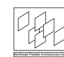
4 Conclusion

This study shows that the recursive bisection technique is a very fast partitioning strategy. Using an object oriented approach one is able to apply the recursive bisection strategy to

- multiple dimensions,
- arbitrary domains on which a linear order is defined for each domain.

The use of MIMD machines makes it possible to increase the speedy of the bisection step. The interface to the algorithm is very simple so that an easy adaption to other problem domains is possible.

Interfaces for two dimensional and three dimensional cartesian domains are existent.



Acknowledgment

I would like to thank Sanjay Ranka for his helpful discussions and initiation of this project.

References

[1] AKL, S. G. *Parallel Sorting Algorithms.* Academic Press, 1985.

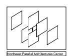
[2] AKL, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, New Jersy, 1989.

[3] FOX, G., JOHNSON, M., LYZENGA, G., OTTO, S., SALMON, J., AND WALKER, D. *Solving Problems on Concurrent Processors.* Prentice Hall, New Jersey, 1988.

[4] KNUTH, D. E. *The Art of Computer Programming, Sorting and Searching*, vol. Volume 3. Addison Wesley, 1973.

[5] VON LASZEWSKI, G. Object Oriented Recursive Bisection on the iPSC860. Tech. Rep. SCCS 477, Northeast Parallel Architectures Center at Syracuse University, April 1993.

[6] VON LASZEWSKI, G. Object Oriented Sorting on MIMD Ring Architectures. Tech. Rep. SCCS 475, Northeast Parallel Architectures Center at Syracuse University, April 1993.



A Sample Code

The interface to the parallel bisection algorithm is quite simple. The user hast to specify a datatype on which a linear order is defined for the relevant dimensions. For the two dimensional plane the datatype has the following structure:

```
typedef struct
  int x;
  int y;
 ITEM;
```

In addition to the definition of the datatype the following functions on this type have to be defined. The function

CopyItem (a,b) – Copies the item a to item b

SwapItem (a,b) – Exchanges the items a and b

RandomItem (a) – Generates a Random Item

PrintItem (me, a) – Prints the item where me specifies the processor id.

COMPARE (a,b,dimension) – compares the two items a and b and returns 0 when a = b, 1 when a < b, and 2 when a > b. The dimension specifies the entry in the data structure ITEM. In the example shown above dimension 0 is equal to x and dimension 1 is equal to y.

The following code gives a complete example for the interface specification in the 2 dimensional plane.

```
#define MaxDimension 2
typedef struct {
  int x;
  int y;
} ITEM;


CopyItem (a,b)
ITEM *a;
ITEM *b;
{
  b→x = a→x;
  b→y = a→y;
}
```

```
SwapItem (a, b)
ITEM *a;
ITEM *b;
{
  ITEM h;
  h.x = a→x; h.y = a→y;
  a→x = b→x; a→y = b→y;
  b→x = h.x; b→y = h.y;
}

RandomItem (a,n)
ITEM *a;
int n;
{
  a→x = Random (n) + 1;
  a→y = Random (n) + 1;
}

void PrintItem (me,a)
int me;
ITEM *a;
{
  printf ("%3d:  %3d %3d", me, a→x, a→y);
}


int Compare_x (a,b)
ITEM *a;
ITEM *b;
{
  if (a→x < b→x) return 1;
  if (a→x > b→x) return 2;
  return 0;
}

int Compare_y (a,b)
ITEM *a;
ITEM *b;
{
  if (a→y < b→y) return 1;
  if (a→y > b→y) return 2;
  return 0;
}


int COMPARE (a,b,dimension)
ITEM *a;
ITEM *b;
int dimension;
{
  switch (dimension) {
  case 0: return Compare_x (a,b); break;
  case 1: return Compare_y (a,b); break;
  otherwise: printf ("COMPARE: dimension wrong <%d>\n",dimension);
  }
}
```