SPRINT: Scalable Partitioning, Refinement, and INcremental partitioning Techniques[1]

Chao-Wei Ou and Sanjay Ranka

4-116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

[1]This work was supported in part by NSF under ASC-9213821 and ARPA under contracts #DABT63-91-C-0028 and NAG-1485. The content of the information does not necessarily reflect the position or the policy of the United State government, and no official endorsement should be inferred.

Abstract

The methods for partitioning computational graphs for practical application can be broadly classified into two classes based on whether physical coordinate information is used for the partitioning. The use of coordinate-based partitioning methods provides faster but lower quality partitioning than methods that use edge information explicitly.

In this paper We describe the SPRINT software for partitioning graphs utilizing only coordinate information on parallel machines.

1 Introduction

From the perspective of parallel computing, a large number of applications can be represented as computational graphs. In a computational graph, the vertices represent tasks that can be executed concurrently, while the edges represent the interaction between them. Informally, the parallelization of these applications requires partitioning the vertices of the graph such that each partition has equal computational load and the cross-edges, which the ends of an edge locate at different partitions, are minimized. The quality of the partitions can be determined by the number of cross-edges, which cause communications in applications. The better the quality of the partitions, the fewer cross-edges there are. Graph partitioning has many important real-world applications, such as domain-decomposition techniques for solving PDEs, laying out circuits in VLSI, etc. In this paper we limit ourselves to the use of graph partitioning for mapping data-parallel applications on a coarse-grained parallel machine.

Exploitation of locality is a necessary feature for the effective parallelization of these applications on parallel machines. The various levels of memory hierarchy present in these architectures include registers, caches, local accesses, non-local accesses, disk accesses, etc. At the very least, parallel machines can be modeled to have two levels of hierarchy, local accesses and non-local accesses. Local accesses are much faster than non-local accesses, typically by at least an order of magnitude.

A number of methods use explicit edge information to achieve very good partitioning. Important heuristics include simulated annealing, mean-field annealing, recursive spectral bisection, recursive spectral multisection, mincut-based methods, and genetic algorithms [1, 10, 11, 12, 13, 18, 20, 24]. Although these heuristics can be applied to arbitrary graphs, empirical evidence of the quality of the partitioning has generally been limited to graphs from two- or three-dimensional physical domains.

SPRINT software focuses on a subclass of applications in which the computational graph is such that the vertices correspond to two- or three-dimensional coordinates, and the interaction between computations is limited to vertices that are physically proximate. Examples of such applications include finite element calculations [21], molecular dynamics [3], particle dynamics [25], particle-in-a-cell [17, 31], region growing [6], and statistical physics [5]. A list of other such applications is given in [4]. For these applications, partitioning can be achieved by exploiting the above property. Essentially proximate points are clustered together and form a partition such that the number of points attached to each partition are approximately equal. Most of the interactions are local and the amount of interprocessor communication is low if proximate points are clustered together. Many such algorithms have been described in the literature, including recursive coordinate bisection [32] and inertial bisection [14]. We have discussed an index-based indexing scheme in [27] and shown that it produces good mappings for computational structures satisfying the above property.

For a large class of irregular and adaptive data-parallel applications [4], the computational structure changes from one phase to another in an incremental fashion. Thus the partitioning information of the previous phase can be effectively utilized to give the partitioning for a new phase. Changes are typically gradual, reflecting adiabatic changes in the physical domain, or large-scale, reflecting additions to a data structure. Molecular dynamics applications often exhibit gradual changes, because interactions between particles are implemented by neighboring lists that change as the atoms move [3]. Adaptive PDE solvers are examples of the large-scale changes. Other examples with which we are familiar include some vision algorithms, including region-growing and

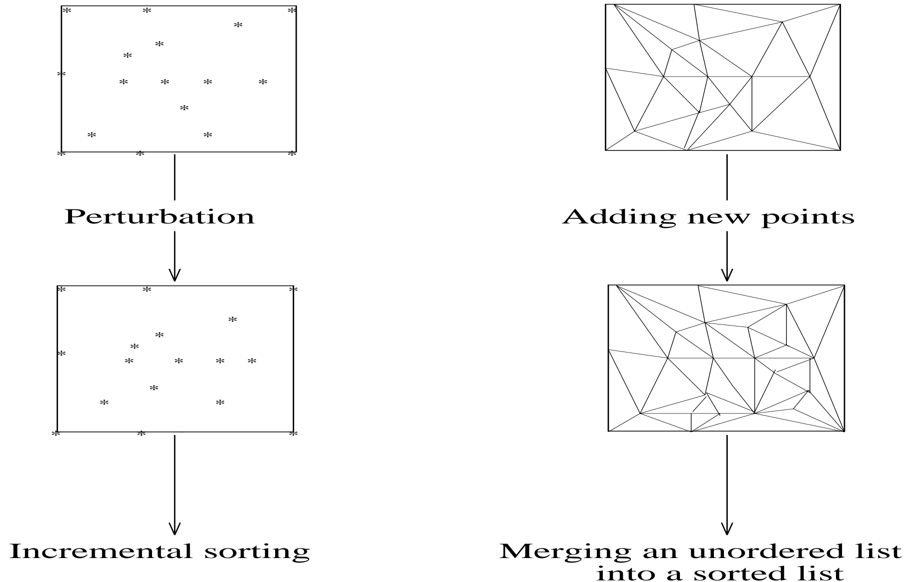

Figure 1: Incremental Aspects

labeling [6], statistical physics simulations near critical points and the particle-sorting phase of a direct Monte Carlo simulation [7]. The key problem in efficient parallelization of these applications is to be able to react quickly to minor modifications in the data structure. The physical and numerical properties of these algorithms typically guarantee that large-scale restructuring of data is needed infrequently. Thus, for effective parallelization, the partitioning of the graph needs to be updated as the graph changes over time. The following scenarios may arise (Figure 1):

- Perturbation: All the coordinates may perturb (within some small distance), e.g., particle-dynamics problems [26].
- Node Additions: New points may be added and/or old points deleted, as with adaptive grids [4].

One option is to repartition the new graph without using previous information. The methods provided in SPRINT allow this to be done in an incremental fashion at a much lower cost than that of other methods.

2 Related research

In recent years several graph-partitioning libraries have been proposed including TOP/DOMEDEC [29] by Charbel Farhat and Host Simon, Chaco [14] by Bruce Hendrickson and Robert Leland, and MeTiS [16] by George Karypis and Vipin Kumar.

TOP/DOMDEC is written in C++ and includes a number of partitioning algorithms (a greedy algorithm [8], RCM-based algorithms [19], principal inertia algorithms [9], recursive graph bisection algorithms, and recursive spectral bisection algorithms [24]). The user interface includes high-speed three-dimensional graphics, an interprocessor communication simulator, and an output function with parallel I/O data structures.

Chaco allows for recursive application of any of several different methods for finding small edge separators in weighted graphs. These methods include inertia-based, spectral-based, Kernighan-Lin, and multilevel methods. Chaco not only provides bisection methods, but also multidimensional methods.

MeTiS is a set of programs for partitioning graphs and for producing fill-reducing orderings for sparse matrices. The algorithms in MeTiS are based on multilevel graph partitioning schemes.

3 SPRINT: Scalable Partitioning, Refinement and INcremental partitioning Techniques

The SPRINT library provides software for parallel graph-partitioning using coordinate information such as index-based partitioning, recursive coordinate bisection, and recursive inertial bisection. It is built on the P4 parallel programming system and allows portability across a wide variety of architectures. As MPI (Message-Passing Interface) becomes more popular, we expect to modify our communication routines to MPI.

```
int     generate_random_graph(n,weight,perm)
int     n, **perm;
double  **x, **weight;
{
int     i,j,lower_global_ref,upper_global_ref,local_n;

lower_global_ref = n*SPRINT_my_proc/SPRINT_num_procs;
upper_global_ref = n*(SPRINT_my_proc+1)/SPRINT_num_procs;
local_n = upper_global_ref-lower_global_ref;

*x = (double *) malloc(local_n*DIM*sizeof(double));
*perm = (int *) malloc(local_n*sizeof(int));
if (weight != NULL) *weight = (double *) malloc(local_n*sizeof(double));;

srand48(SPRINT_my_proc);
for (i=0;i<local_n;i++) {
    for (j=0;j<DIM;j++)
        x[i*dim+j] = drand48();
    perm[i] = i+lower_global_ref;
    if (weight != NULL) *weight[i] = NNODES*drand48();
}
return(local_n);
}
```

Figure 2: A simple function to generate a random graph.

Figure 2 describes a simple function to generate a random graph. This example is included to show the basic data structures and metadata used in SPRINT to describe graphs. **lower_global_ref** and **upper_global_ref** specify the boundaries of the graph in each processor; **local_n** represents

the number of vertices in a processor; **x** represents the double-precision coordinates of vertices in one-dimensional array; **perm** is the global reference for vertices. Currently, SPRINT supports double-precision coordinates and weights of vertices.

```
#include <stdio.h>
#include "P4_HOME_DIR/include/p4.h"
#include "SPRINT_HOME_DIR/include/sprint_defs.h"
#include "SPRINT_HOME_DIR/include/sprint_funcs.h"

#define NNODES 20 /* number of vertices of the graph */
#define DIM 2 /* number of dimensions of the graph */

void    main(argc,argv)
int     argc;
char    **argv;
{
int local_n;
double *x;
int *perm;

p4_initenv(&argc,argv);
SPRINT_init(0,0,1);

local_n = genertae_random_graph(NNODES,&x,NULL,&perm);

SPRINT_coordinate_bisection(DIM,local_n,&x,SPRINT_DBL,&perm);

free(x);
free(perm);
SPRINT_done();
p4_wait_for_end();
}
```

Figure 3: A simple example demonstrating the use of the SPRINT library.

The example in Figure 3 shows a simple code that uses SPRINT software. The code generates a random graph and partitions this graph into the desired number of partitions. The function **p4_initenv** reads the argument list that contains the information needed in P4 (includes process creation). **SPRINT_init** initializes the working space used for SPRINT. **generate_rendom_graph** generates a random graph for partitioning. The partitioning function **SPRINT_coordinate_bisection** partitions the random graph into the desired number of partitions. The space can be de-allocated by using **SPRINT_done** and **p4_wait_for_end**. Examples throughout the paper that illustrate the functionality of SPRINT will use similar data structures.

4 Coordinate-Based Partitioners

The computational graphs considered in SPRINT assume that most interactions occur between vertices that are physically proximate in two or more dimensions. Quality comparisons of partitioning achieved by some of these methods such as index-based partitioning and recursive orthogonal bisection with edge-based partitioners (e.g., recursive spectral bisection) can be found in our previous work [23]. In the following we briefly describe the various methods and software interfaces.

4.1 Index-Based Partitioning (IBP)

z-Curve indexing and the Hilbert space-filling curve are two of several ways to index pixels in a two-dimensional grid. These two indexing schemes are shown in Figure 4 (a) and Figure 4 (b) for a graph in which the set of vertices are arranged in an 8×8 grid. z-Curve and Hilbert space-filling curve indexing maintain the property that close vertices have close indices along both dimensions. These indexing schemes can be generalized to n-dimensions and used to convert an n-dimensional index into a one-dimensional index such that proximity in the n-dimensions is generally maintained. Index-based algorithms for partitioning graphs have been described in [23]. An IBP algorithm includes five phases—indexing, sample sorting, distributing non-local indices, balancing, and updating the reference table.

```
42 43 46 47 58 59 62 63        21 22 25 26 37 38 41 42
40 41 44 45 56 57 60 61        20 23 24 27 36 39 40 43
34 35 38 39 50 51 54 55        19 18 29 28 35 34 45 44
32 33 36 37 48 49 52 53        16 17 30 31 32 33 46 47
10 11 14 15 26 27 30 31        15 12 11 10 53 52 51 48
08 09 12 13 24 25 28 29        14 13 08 09 54 55 50 49
02 03 06 07 18 19 22 23        01 02 07 06 57 56 61 62
00 01 04 05 16 17 20 21        00 03 04 05 58 59 60 63

          (a)                            (b)
```

Figure 4: Different indexing schemes for an 8×8 image: (a) z-curve and (b) Hilbert space filling curve.

The z-Curve index can be derived easily by interleaving the indices. A simple example of interleaving indices is as follows. Suppose $index_1 = 001$, $index_2 = 010$, and $index_3 = 110$. Then the interleaved index would be 001011100. In the above case the number of bits in each dimension are equal. This could easily be generalized to cases where the sizes are different. For example, if $index_1 = 101$, $index_2 = 01$, and $index_3 = 0$, then the interleaved index would be 100110. This is done by choosing bits (right to left) of each dimension one by one, starting from dimension 3. When the bits of a particular dimension are no longer available, that dimension is not considered.

Another indexing scheme that maintains proximity in multiple dimensions is based on Hilbert space-filling curves [15] (Figure 4 (c)). We have performed experiments with this indexing, and the quality of partition obtained is similar to the z-Curve indexing. However, most of the algorithms discussed in this paper are independent of the indexing method used.

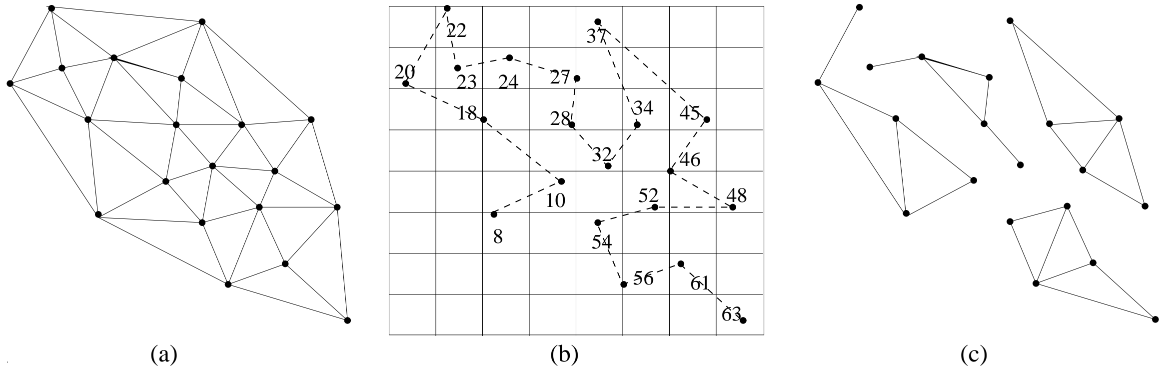

Figure 5: An example of partitioning the graph into 4 partitions by IBP using Hilbert curve: (a) The initial graph; (b) indices for all vertices after indexing using Hilbert space-filling curve; (c) the partitioned graph. The index assignment is based on an 8×8 image.

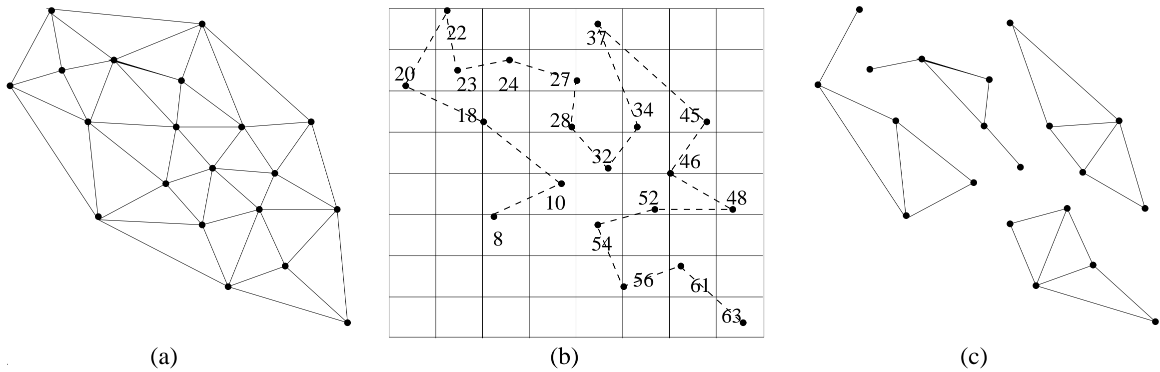

Figure 6: An example of partitioning the graph into 4 partitions by IBP using z-curve: (a) The initial graph; (b) indices for all vertices after indexing using z-curve space-filling curve; (c) the partitioned graph. The index assignment is based on an 8×8 image.

After indexing is done, an efficient sorting algorithm can be applied to sort these vertices according to their indices. We used sample-based sorting methods [2, 28, 30]. First, some elements are randomly picked from the distributed list and a parallel bitonic sort or parallel merging sort is performed to sort the sampling elements. Second, the sorted sampling list is divided into p equal parts and the maximum elements in each partial list are recorded to form boundaries for all processors. Finally, all elements are distributed among the processors according to the previous boundaries. The elements in the processor with the lower identification number are fewer than the elements in the processor with the higher identification number. Two examples of graphs are shown in Figure 5 and Figure 6 for Hilbert space-filling curves and z-curve space-filling curve, respectively. An index-based partitioning scheme does not need to sort the local elements, quickly finding the m smallest or largest elements instead. The algorithm used in SPRINT is based on quick selection to find the desired elements and is followed by a communication phase to balance the elements.

The global-distributed reference table needs to be updated for applications that require the new location of the node. The syntax of the procedures in SPRINT is given in Figure 7. These four sets of functions perform the parallel index-based partitioning using table-look-up for the input graph. **_hilbert** and **_zcurve** use the Hilbert space-filling curve and z-Curve, respectively. If the input graph is non-uniformly weighted, the suffix with "_w" will take the weights into account to partition the graph. If the attributes of vertices in the graph are coordinates and/or weights, users may move these attributes along with the partitioner by calling the functions with the suffix "_move". These functions allow users to acquire the data that has been located at the proper processors after the completion of the function. All four of these sets of functions require that users assign a globally unique integer reference to each vertex. The index will be used to create the global-distributed reference table. The range of global references is $[0 \,..\, N)$ where N represents the number of vertices of the graph. All these functions return the new number of vertices after partitioning. An example for the correct function call sequence in SPRINT is given in Figure 8.

4.2 Recursive Coordinate Bisection (RCB)

Recursive coordinate bisection orthogonally bisects the graph along the largest dimension into two almost-equal weighted subgraphs and continuously and recursively bisects these two subgraphs along the largest dimension of each subgraph into two subsubgraphs in each subgraph. The key feature of this scheme is to determine the median vertex, which can be quickly found at the current dimension. The common usage of recursive coordinate bisection is to find the median vertex recursively. The median finding divides the graph into two subgraphs based on the geometry of the graph and calculates the number of vertices or the weight for both subgraphs. The subgraph with the larger number of vertices or greater weight will be divided into another two subgraphs until the median vertex is found. If the graph contains some vertices with the same coordinates at the current dimension, the tie can be broken by arbitrarily or randomly assigning those vertices to one of two subgraphs.

In parallel implementation RCB needs one all-to-all communication in each recursive median-finding step. This fact would cause RCB to spend more time in finding the median vertex. The improvement can be made by creating sampling buckets to reduce the number of recursive steps. The number of sampling buckets can be defined as the bandwidth and the packet size of the communication. This would speed up the recursive median-finding process to the logarithm of the number of sampling buckets based on 2. Figure 9 shows the quick-median finding based on

int SPRINT_ibp_hilbert(ndims, local_n, x, xtype, perm)
int SPRINT_ibp_zcurve(ndims, local_n, x, xtype, perm)

int SPRINT_ibp_hilbert_w(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_ibp_zcurve_w(ndims, local_n, x, xtype, wgt, wtype, perm)

int SPRINT_ibp_hilbert_move(ndims, local_n, x, xtype, perm)
int SPRINT_ibp_zcurve_move(ndims, local_n, x, xtype, perm)

int SPRINT_ibp_hilbert_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_ibp_zcurve_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of vertices in the processor.
void ***x*: input coordinates of vertices in one dimensional row-major representation. It returns the new pointer of coordinates of the local vertices.
int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input weights of vertices and return the new pointer of weights of the local vertices.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of the global references and return the new pointer of the global references of the local vertices.

Figure 7: Syntax of routines for the index-based partitioning

```
#include        <stdio.h>
#include        "P4_HOME_DIR/include/p4.h"
#include        "SPRINT_HOME_DIR/include/sprint_defs.h"
#include        "SPRINT_HOME_DIR/include/sprint_funcs.h"

#define NNODES 20       /* number of vertices of the graph */
#define DIM 2           /* number of dimensions of the graph */

void    main(argc,argv)
int     argc;
char    **argv;
{
int     local_n, new_local_n;
double  *x, *wgt;
int     *perm;

p4_initenv(&argc,argv);
SPRINT_init(0,0,1);

local_n = genertae_random_graph(NNODES,&x,&wgt,&perm);

new_local_n = SPRINT_ibp_hilbert_move_w(DIM,local_n,&x,SPRINT_DBL,&wgt,SPRINT_DBL,&perm);

free(x);
free(perm);
free(wgt);
SPRINT_done();
p4_wait_for_end();
}
```

Figure 8: An example of using the index-based partitioner based on the Hilbert space-filling curve to redistribute weighted graphs.

8 buckets. The implementation of RCB in SPRINT focuses on quickly finding the close-median vertex instead of the exact median vertex. Experiments show that the quality of partitioning remains good. Figure 10 shows an example of partitioning a graph into 4 partitions.

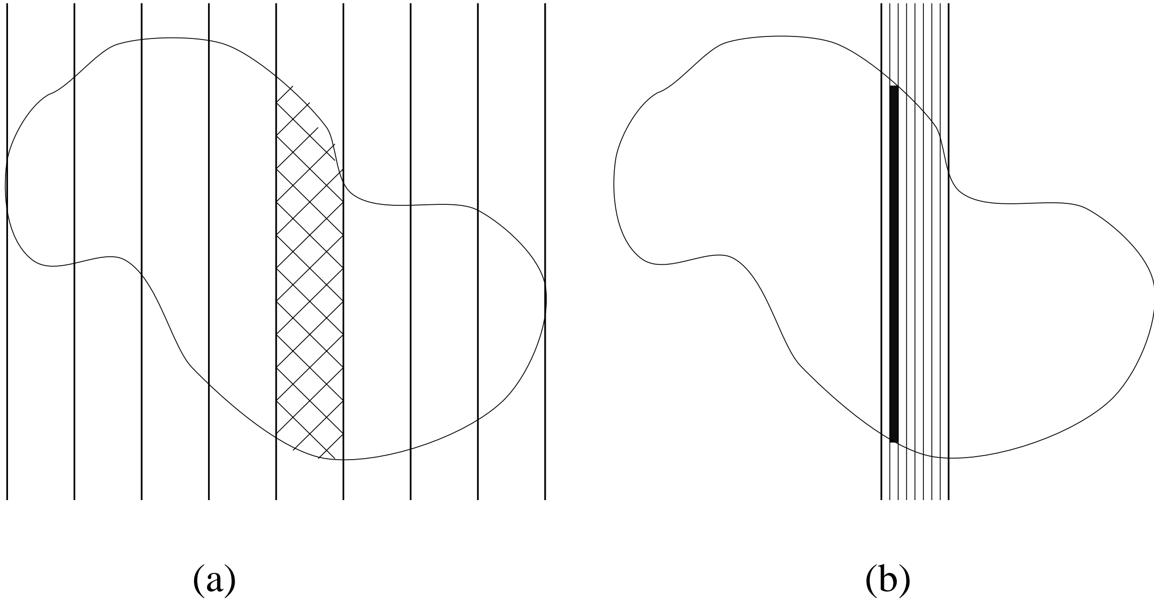

Figure 9: Quick integer median-finding.

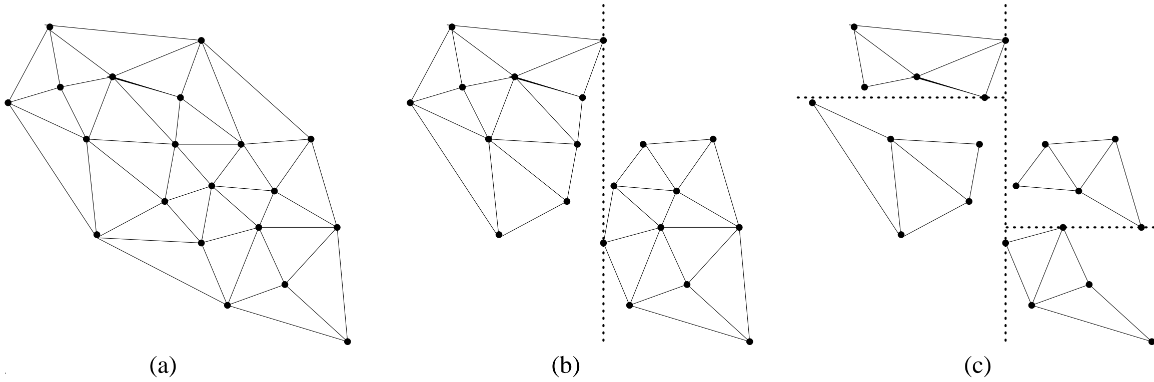

Figure 10: An example of partitioning a graph into 4 partitions by RCB: (a) The initial graph; (b) partitions of the first recursive step; (c) the partitioned graph after RCB

Recursive Inertial Bisection (RIB)

Recursive inertial bisection bisects the graph along the longest extension into two subgraphs to avoid long and thin subgraphs. RIB first calculates and diagonalizes the initial tensor of centroid (the center of mass) to find the principle axis, which is the longest extension. All the vertices project to this axis to form a linear vector. This vector is bisected at the median to two sub-vectors, such that the element values in the lower vector are less than those in the higher vector. According to these two sub-vectors, two subgraphs can be formed. The same process is applied to the subgraphs recursively until the desired number of partitions is achieved.

Finding the principle axis that corresponds to solving the eigen value of the tensor. In the RIB problem it calculate the eigen vector corresponding to the largest eigen value. In parallel implementation this step can be done synchronously, since the size of the tensor is small (corresponding to the number of dimensions). The method for finding the median of the vector is the same as that used in recursive coordinate bisection. Figure 11 shows the partitioning of a graph into 4 partitions.

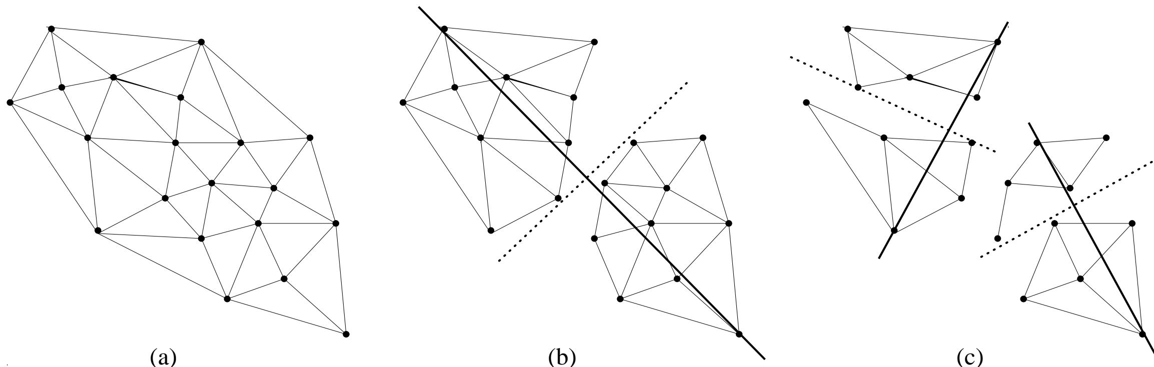

Figure 11: Partitioning the graph into 4 partitions by RIB: (a) The initial graph; (b) partitions of the first recursive step (c) the partitioned graph after RIB. The solid-thick lines represent the principle axis.

The syntax of the routines for coordinate bisection and inertial bisection are given in Figure 12. These four functions perform recursive coordinate bisection. The functions with the suffix "_w" or "_move" retain the same meanings as those in the previous sections. The total cost may be reduced if one moves the partial attributes of vertices to the proper processor and redistributes the rest of the attributes by using **SPRINT_data_redistribute**, because this method needs coordinates to compute the momentum of the graph and subgraphs. Coordinates have to be moved after each recursive step of coordinate-based bisection, as do the weights.

5 Incremental Index-Based Partitioning – Perturbation

In applications such as molecular dynamics, particle-in-a-cell methods, particle dynamics, etc., the interaction between several particles is simulated. These particles are dispersed in a two- or three-dimensional space, and the simulation is performed for a large number of time steps. At each time step, the numerical approximation techniques used for simulation dictate that the amount of particle movement be small. Further, most of the important interactions in the simulation are limited to points that are physically proximate. Assume that index-based mapping is used to partition these points. The corresponding index is expected to change by a small amount, but this is not always the case (e.g., the index of (3, 3) is 15, while the index for (3, 4) is 26).

Thus the incremental mapping for perturbation of the particles, after a few time steps, can be reduced to the following problem. There is a globally sorted list A of size n. A nearly globally sorted list B is derived from list A by perturbing each element by a small amount (on an average) by adding or subtracting a small random number.

We have developed an algorithm to distribute the elements in B efficiently by utilizing information of the previously globally sorted list A [22]. The basic principle used by the algorithm is the fact that A is close to B and globally sorted, and the partitions of A can be used to distribute

int SPRINT_coordinate_bisection(ndims, local_n, x, xtype, perm)
int SPRINT_inertial_bisection(ndims, local_n, x, xtype, perm)

int SPRINT_coordinate_bisection_w(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_inertial_bisection_w(ndims, local_n, x, xtype, wgt, wtype, perm)

int SPRINT_coordinate_bisection_move(ndims, local_n, x, xtype, perm)
int SPRINT_inertial_bisection_move(ndims, local_n, x, xtype, perm)

int SPRINT_coordinate_bisection_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_inertial_bisection_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of local vertices in the processor.
void ***x*: input the pointer of coordinates of local vertices and return the new pointer of coordinates of local vertices after partitioning.
int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input the pointer of weights of local vertices and return the new pointer of weights of local vertices after partitioning.
void ***wgt*: input pointer of weights of vertices.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of global references of local vertices and return the new pointer of the global references of local vertices after partitioning.

Figure 12: Syntax of routines for recursive-coordinate bisection and recursive-inertial bisection.

the elements of B into lists of approximately equal size. Each of these lists is such that all the elements of the list are greater than the previous list. An incremental globally sorting algorithm assumes the presence of a globally sorted list A, of size n, divided equally among all the processors in a contiguous fashion. The list A is divided into p partitions to get approximate boundaries of p partitions of B on each processor. The maximum element of the local list A is used to create an array of the size of the number of partitions in each processor; $Boundary_i[j] := A_i[\frac{jn}{p}]$, $1 \leq j \leq p$. For each element of B an appropriate partition must be obtained.

Each element of B can be classified into three categories, depending on whether the element belongs to the same partition as the corresponding element of A or to another partition. Those elements that are non-local to the current partition will be moved to the proper partition corresponding to the previous boundary information. The quick finding algorithm will then be applied to the local list, followed by the balancing step. After all these steps, list B will be globally sorted.

The syntax of incremental index-based partitioning for perturbation is given in Figure 13.

Description

Four sets of functions perform the incremental index-based partitioning for the applications that graphs slightly change after each computation iteration. The suffix "_w" and "_move" represent the same meanings in the section of Index-Based Partitioning. Uses of the incremental index-based partitioning are restricted to calling the incremental partitioning functions in SPRINT instead of developing the communication functions themselves, since the the global reference information will not be maintained for the next calling routine. If communication functions to move data are needed, static partitioners may be used instead. Since the cost of an incremental partitioner is much less than the cost of a static partitioner, users may lose this advantage.

6 Incremental Index-Based Partitioning – Addition

For many applications, such as adaptive meshes, new vertices are added to the computational graph. Typically, this is done in a localized area to study numerical behavior more precisely. These refinements are based on the solution of the previous phase and are available only at runtime. During a typical simulation vertices may be added in a particular portion, only to be removed after a few phases. The following discussion is limited to the case when vertices are added to the computational graph. All of these algorithms can be easily modified when this is not the case.

Remapping requires calculating the z-Curve indices or Hilbert space-filling curve indices of the new vertices, which must be combined with the indices of the previous phase. Since the previous mapping is available, this corresponds to adding an unsorted list of integers (corresponding to the indices of the new vertices that are added) to a globally sorted list (corresponding to the indices of the old vertices).

Let A represent a sorted list of n integers, and let B represent an unsorted list of m integers. A simple sequential approach for merging list B into list A is to sort B, followed by merging the two sorted lists. We assume that the list A is already globally sorted and divided equally among all the processors. This corresponds to the partitioning of the previous phase. The new vertices added in the new phase are assumed to be generated locally. This is not always going to be the case and, fact, for most practical cases the incremental vertices are added in localized portions.

int SPRINT_perturb_ibp_hilbert(ndims, local_n, x, xtype, perm)
int SPRINT_perturb_ibp_zcurve(ndims, local_n, x, xtype, perm)

int SPRINT_perturb_ibp_hilbert_w(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_perturb_ibp_zcurve_w(ndims, local_n, x, xtype, wgt, wtype, perm)

int SPRINT_perturb_ibp_hilbert_move(ndims, local_n, x, xtype, perm)
int SPRINT_perturb_ibp_zcurve_move(ndims, local_n, x, xtype, perm)

int SPRINT_perturb_ibp_hilbert_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_perturb_ibp_zcurve_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of vertices in the processor.
void ***x*: input the pointer of coordinates of the local vertices in one-dimensional row-major representations and return the new pointer of coordinates of the local vertices after partitioning.
int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input the pointer of weights of local vertices and return the new pointer of weights of local vertices after partitioning.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of the global references and return the new pointer of global references of the local vertices.

Figure 13: Syntax of routines for incremental index-based partitioning for perturbation.

This typically correspond to all the new elements belonging to a few processors.

According to the previous assumption, the new added vertices need to be distributed to the proper processor corresponding to the boundary information provided by the previous partitioning phase. Since most new vertices reside in the local partition, the load may be imbalanced while performing the quick finding. This would, however, save the cost of the extra communication and the total cost of the incremental partitioner would be much less than that of applying the static partitioner from scratch when the communication cost vs. computation cost and the number of additional vertices is a fraction of the number of vertices of the graph.

The syntax of incremental index-based partitioning for addition is given in Figure 14. These four sets of functions perform incremental index-based partitioning for adding new vertices to the graph. Functions with the suffix "_w" or "_move" retain the same meanings as mentioned in the previous sections. The same restrictions as in the previous section (Incremental Index-Based Partitioning – Perturbation) are applied to these four sets of functions. The advantages of these functions are that the costs are much less than the costs of the static partitioners, since most of the computations and communications occur in the additional vertices.

7 Incremental Index-Based Partitioning – Deletion

The syntax of the incremental index-based partitioning for deletion is in Figure 15. These functions need global references instead of graph data because the attributes of the vertices remain the same. The only argument the user is asked to input is the global references of the rest of the graph's vertices. SPRINT does not provide the options as did the previous partitioning functions, as the attributes of vertices are not involved in these functions. The data redistributing function in the "Utilities" section can be used to rearrange the vertices.

8 Other Utilities

8.1 Initialization

This function is used to set up the following global variables used in SPRINT:

- *SPRINT_num_procs* is an integer to indicate the total number of processors involved in SPRINT.
- *SPRINT_my_proc* is an integer to indicate the processor identification number of the processor.
- *SPRINT_logic_id* is an integer to indicate the processor identification number of the current processor group, especially for recursive coordinate bisection and recursive inertial bisection.
- *SPRINT_global_max* is a double-precision array of the size of dimensions of the graph to store the global maximum coordinates in each dimension. In other words, this will provide the upper bound for the input graph to all partitioning functions in the SPRINT.
- *SPRINT_global_min* is a double precision array of the size of dimensions of the graph to store the minimum coordinates in each dimension and this corresponds to lower bound for the graph.

____ **int SPRINT_inc_ibp_hilbert**(ndims, local_n, perm, add_local_n, add_x, xtype, add_perm)
int SPRINT_inc_ibp_zcurve(ndims, local_n, perm, add_local_n, add_x, xtype, add_perm)

int *local_n*: input the number of old vertices in the processor.
int SPRINT_inc_ibp_hilbert_w(ndims, local_n, perm, add_local_n, add_x, xtype, add_wgt, wtype, add_perm)
int SPRINT_inc_ibp_zcurve_w(ndims, local_n, perm, add_local_n, add_x, xtype, add_wgt, wtype, add_perm)

int SPRINT_inc_ibp_hilbert_move(ndims, local_n, x, xtype, perm, add_local_n, add_x, add_perm)
int SPRINT_inc_ibp_zcurve_move(ndims, local_n, x, xtype, perm, add_local_n, add_x, add_perm)

int SPRINT_inc_ibp_hilbert_w_move(ndims, local_n, x, xtype, wgt, wtype, perm, add_local_n, add_x, xtype, add_wgt, wtype, add_perm)
int SPRINT_inc_ibp_zcurve_w_move(ndims, local_n, x, xtype, wgt, wtype, perm, add_local_n, add_x, xtype, add_wgt, wtype, add_perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of old vertices in the processor. void ***x*: input the pointer of coordinates of the old local vertices and return the new pointer of coordinates of the old local vertices and the additional vertices.
int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input the pointer of weights of the old local vertices and return the new pointer of weights of the old local vertices and the additional vertices.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of the global references of the old local vertices and return the new pointer of the global reference indices of the old vertices and new vertices.
int *add_local_n*: input the number of local additional vertices in the processor.
double **add_x*: input the coordinates of additional vertices in one dimensional row-major representations.
double **add_wgt*: input the weighted of local additional vertices.
int *add_perm*: input the global references of additional vertices.

Figure 14: Syntax of routines for incremental index-based partitioning for addition.

int SPRINT_del_ibp(local_n, perm)
int SPRINT_del_ibp_w(local_n, perm)

int *local_n*: input rest number of local vertices after deletion.
int ****perm***: input the pointer of the global references of the rest vertices and return the new pointer of the global reference indices of the local vertices.

Figure 15: Syntax of routines for incremental index-based partitioning for deletion.

---

void SPRINT_init(debug_flag, demo_flag, log_flag)

int *debug_flag*: turn on the debuging process to track the current SPRINT status if the flag is set to 1; otherwise set the flag to 0. This option will help users to check the status of communications and generate the detail timing inside the set of index-based partitioners.
int *demo_flag*: turn on the demonstrating process to allow users check the current information of partitions if the flag is set to 1; otherwise set the flag to 0.
int *log_flag*: turn on the script option if the flag is set to 1; otherwise set the flag to 0. This will generate a script file for each processor to record the intremedia output for debuging or demonstrating the current partitions.

- *SPRINT_dim* is an integer of the number of useful dimensions used in "Indexing". Most of the time this value will equal to the dimensions of the graph. The exception occurs when one or more dimensions are too small as compared to the most significant dimension.
- *SPRINT_vtx_boundary* is an integer array to store the boundaries of the number of vertices for all processors.
- *SPRINT_idx_boundary* is an integer array to stored the largest index in each processor for the usage of incremental IBP for additional vertices and perturbation. This will help SPRINT locate the new vertices to the proper processors.
- *SPRINT_ref_boundary* is an integer array to store the boundary entries of the reference table.
- *SPRINT_ibp_old_idx* is a structural array to store the index values for each local vertices acquired from the previous partitioning. This is used in incremental IBP for additional vertices, perturbation, and deleting vertices. Each element of this array contains two attributes, which are the index generated from the index-based partitioner and the global reference.
- *SPRINT_ibp_old_idx_w* is a structural array to store the index values for each local vertices acquired from the previous partitioning. This is used in incremental IBP for additional vertices, perturbation, and deleting vertices. Each element of this array contains three attributes: the index generated from the index-based partitioner, the weight, and the global reference.
- *SPRINT_ref_table* is a structure array containing two attributes, the processor ID and the local address, to indicate the location of the physical data (Figure 16).

8.2 Termination

The following function will free the space of those global variables used in SPRINT.
void SPRINT_done()

8.3 Data Redistribution

---

int SPRINT_data_redistribute(local_n, data, size)

int *local_n*: input the number of local vertices in the processor. char ***data*: input the pointer of local data associated with the global references and return the new pointer of local data after the partitioning. Data can be in any type of array.
int *size*: input the size of an element in the data.

int SPRINT_data_redistribute_v(local_n, data, size)

int *local_n*: input the number of local vertices in the processor. char ***data*: input the pointer of local data associated with the global references and return the new pointer of local data after the partitioning. Data can be in any type of array.
int ***size*: input the pointer of the size of each element in the data and returned the new pointer of the size of each element in the redistributed data.

SPRINT_data_redistribute will redistribute the graph to the processors based on the mapping returned from partitioning functions. **SPRINT_data_redistribute_v** will redistribute variable sized graphs. Both functions return the number of local vertices in the processor after partitioning. A better data structure that users may think about before implementing the applications might be a structural type in the C language because users have to use this function to redistribute each attribute of the graph if the data type is an array, or a lot of local data copies if the data type is a pointer.

8.4 Communication Schedule

---

void SPRINT_create_schedule(local_n, ref_list, local_ref_list)

int *local_n*: input the number of local vertices in the processor after data redistribution.
int **ref_list*: input an array of global references that are needed to perform the local computations.
int **local_ref_list*: output an array of local references corresponding to global references.

This function will create the communication schedule and rearrange the collection of local and non-local references to the local references. For some applications that need to compute the attributes of neighbor vertices for each vertex in a iteration, users have to specify the reference list (adjacent list) for the input of this function and this function will return a new reference list that

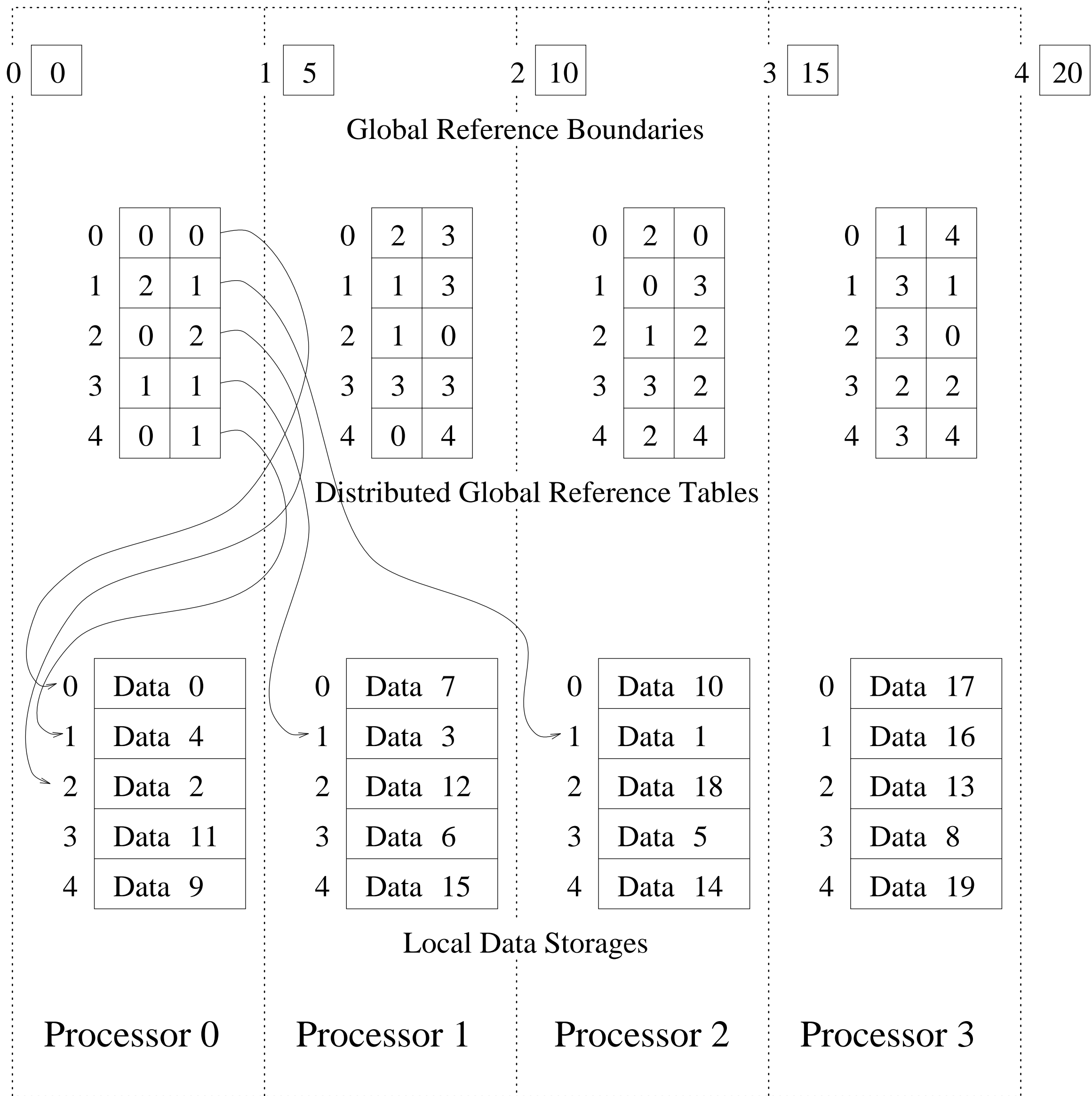

Figure 16: An example of distributed global reference tables on 4 processors: The first row represents the boundaries of the reference table for all processors. The second row shows that the distributed reference tables point to the local data storages where the first column represents the processor ID, and it represents the local address in that processor. The third row represents the local data storages on all processors.

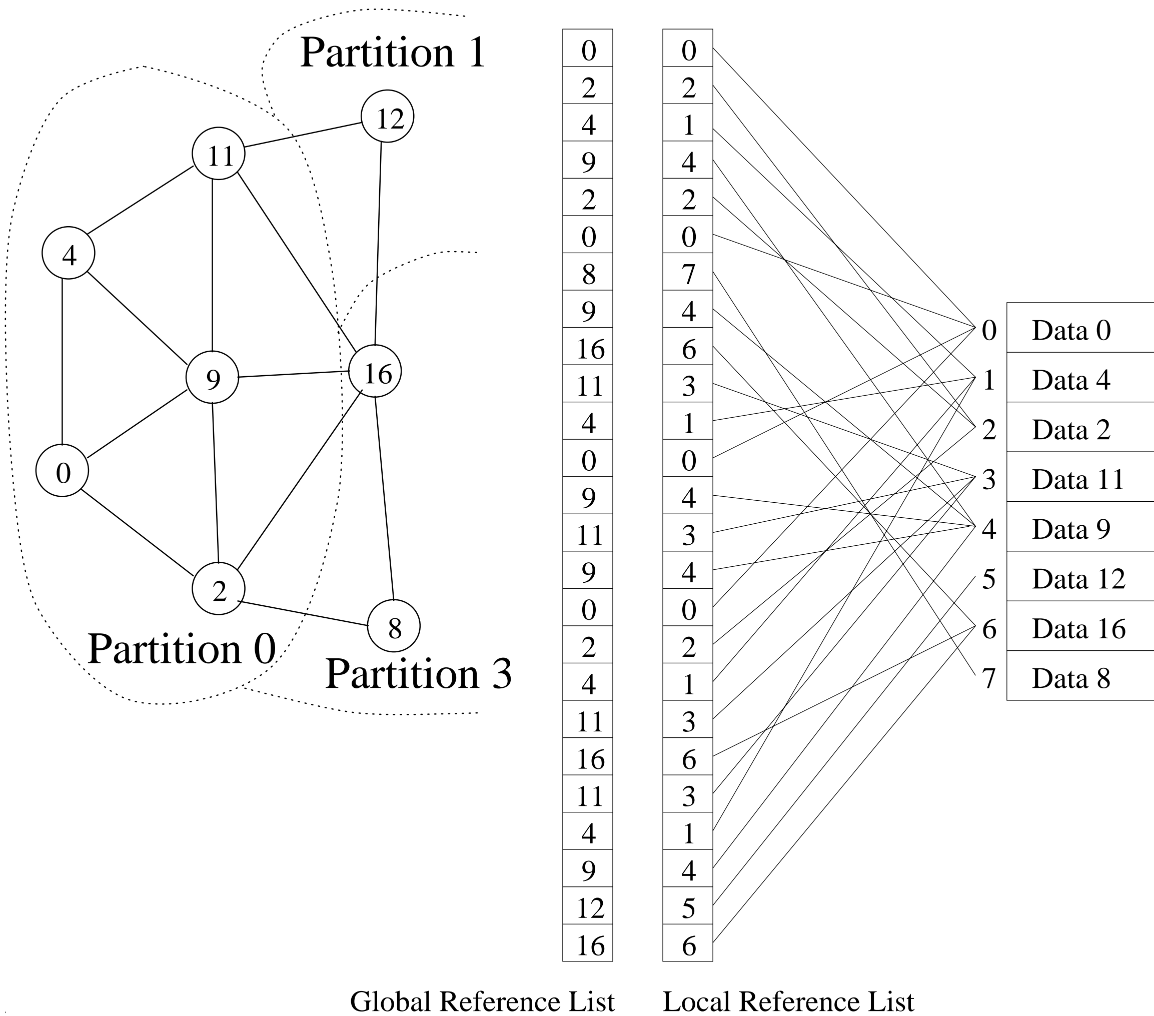

Figure 17: A mapping from the global references to local references for the example graph at the left.

points to the data stored locally. This function will remove the duplicate references by using the hash table to reduce the size of messages and the size of local storage. Figure 17 describes the mapping of global references to local references.

8.5 Data Localization

---

void SPRINT_gather(data, outdata, size)

char **data*: input the data of local vertices in character type.
char **outdata*: output the data of local referencing vertices in character type.
int *size*: input the size of the element in the data.

This function provides the communication to gather those data that are non-local to the processor

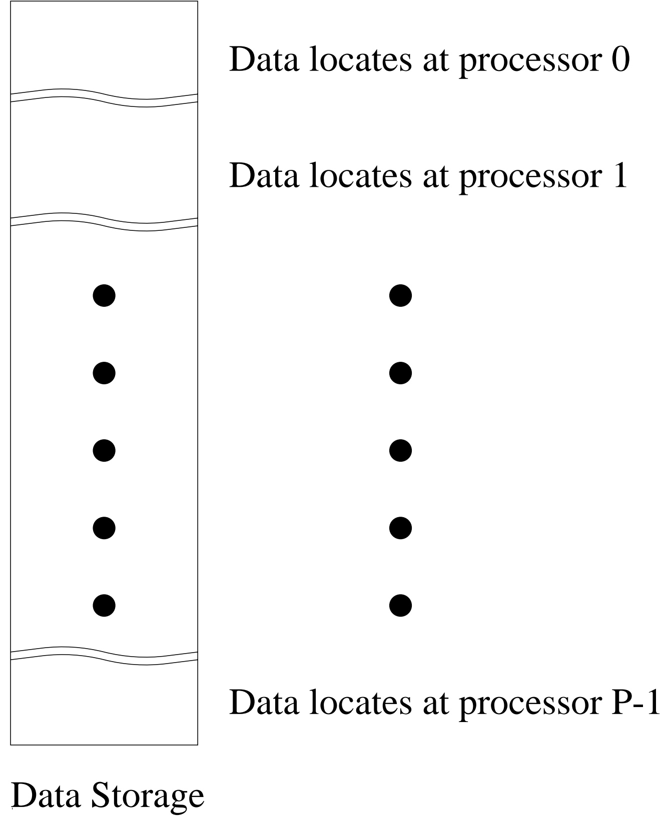

Figure 18: Storage arrangement for local and non-local data after localizing data.

Graph ID	Graph 1	Graph 2	Graph 3	Graph 4	Graph 5	Graph 6	Graph 7	Graph 8
Graph Name	Hscts	Aircraft	Kall1	Barth4	10k	Barth5	Vaughan	50k
$\|V\|$	2028	2851	4363	6019	9428	15606	29681	53961
$\|E\|$	20341	15093	26570	17473	59863	45878	81795	343576

Table 1: Graphs used in experiments

and rearrange the localized data corresponding to the schedule created by **SPRINT_create_schedule**. Users may be asked to pack the data in a one-dimensional array and indicate the size of the element of the data array to this function. After data localization, users can access the localized data by using *local_ref_list*. Figure 18 shows the locations of all data that will be used in the next computation iterations. The data is arranged corresponding to the processor ID of the data in ascending order.

9 Performance

In this section Table 1 gives the graphs used in our experiments, and Figure 19 and 20 present the results on the CM5 and the PARAGON, respectively.

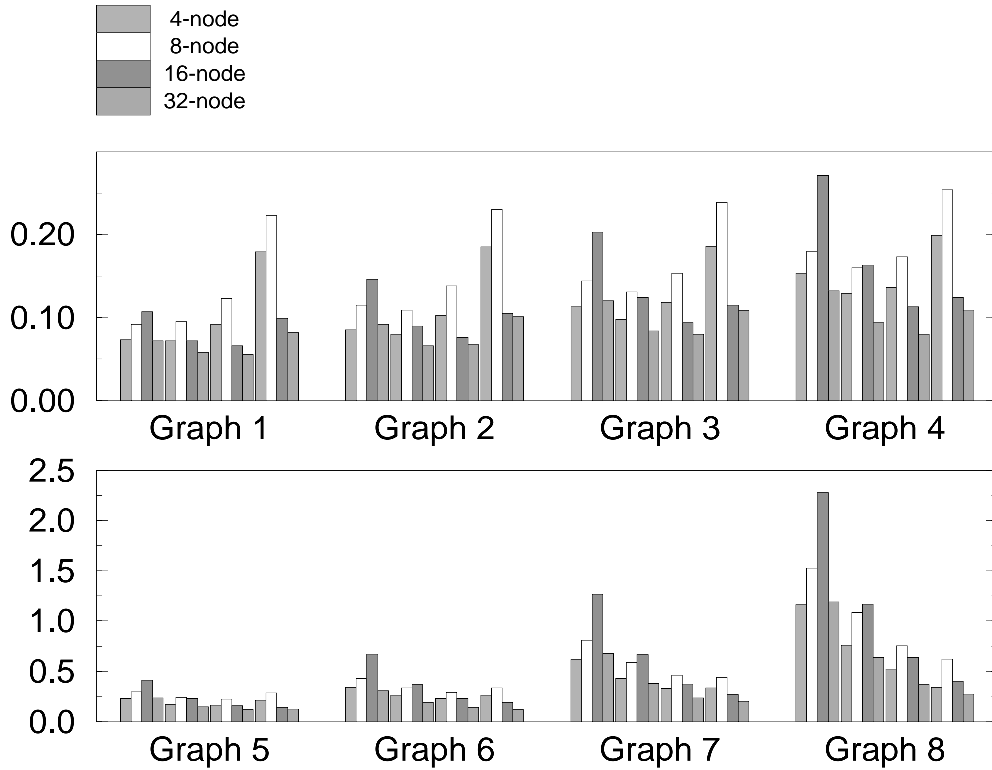

Figure 19: Partitioning time for Graphs on the CM-5.

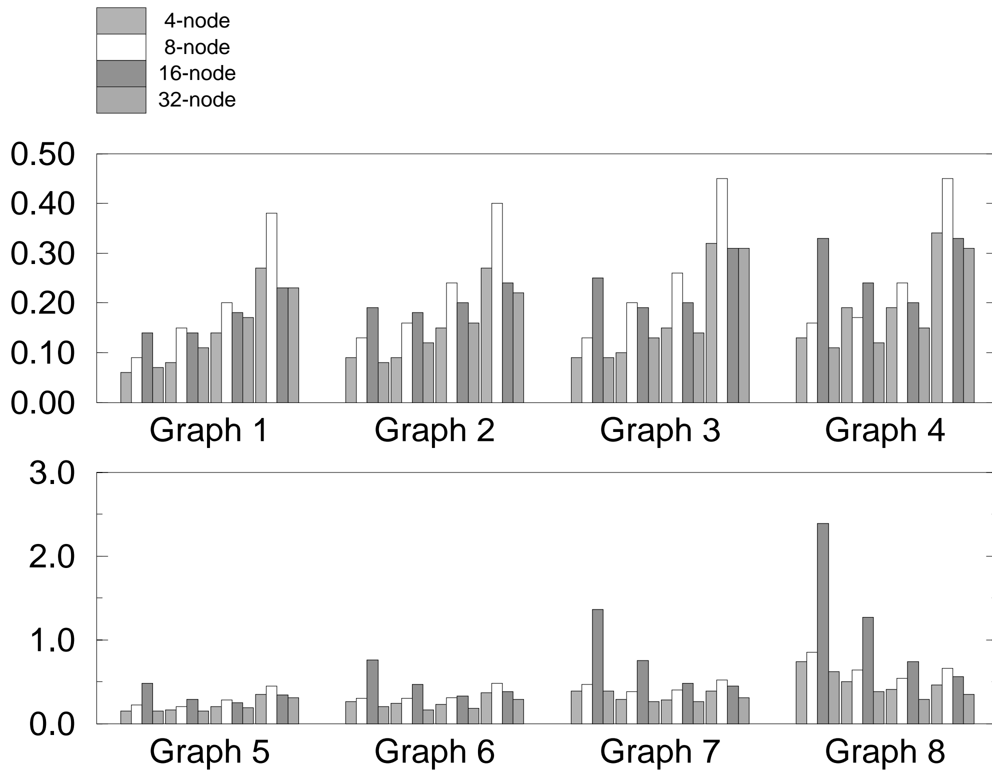

Figure 20: Partitioning time for Graphs on the PARAGON.

10 Conclusions

The performances of the four partitioning schemes included in the SPRINT package are close in quality, but index-based partitioning provides more flexibility for incremental aspects. SPRINT not only provides partitioners, but also a parallel computation environment for developing parallel applications on message-passing-based machines. Users can easily develop the parallel applications on SPRINT. We believe that SPRINT is useful for those who work on parallel computing applications.

Acknowledgement

Authors would like to thank Geoffrey Fox, the director of Northeast Parallel Architectures Center, for his insightful discussion and supports, Ibraheem Al-Furaih for providing experimental results from the "Chaco" partitioning package, Robert Leland and Bruce Hendrickson for providing the Chaco software package, and Elaine Weinman for editing the paper.

References

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

[2] M. Bolorforoush, N. Coleman, D. Quammen, and P. Wang. A Parallel Randomized Sorting Algorithm. *Proceedings of the International Conference on Parallel Processing*, August 1992.

[3] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus. A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4:187, 1983.

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

[5] P. Coddington and C. Baillie. Cluster Algorithms for Spin Models on MIMD Parallel Computers. *Proceedings of the 5th Distributed Memory Computing Conference*, pages 384–388, Charleston, SC, April 1990.

[6] N. Copty, S. Ranka, G. Fox, and R. Shankar. SIMD and MIMD region growing algorithms on the CM-5. *International Conference on Parallel Processing*, 1994.

[7] L. Dagum. Data Parallel Sorting for Particle Simulation. *Concurrency*, 4:241–255, May 1992.

[8] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comp. and Struct.*, 28:579–602, 1988.

[9] C. Farhat and M. Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *International J. Numer. Meth. Engrg.*, 36:745–764, 1993.

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

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

[12] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. Technical report, Sandia National Laboratories, Albuquerque, NM 87185, 1992.

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

[14] B. Hendrickson and R. Leland. The Chaco User's Guide, Version 1.0. Technical report, Sandia National Laboratories, October 1993.

[15] D. Hilbert. Uber die steitige Abbildung einer linie auf ein Flachenstuck. *Math. Ann, 38*, 1891.

[16] G. Karypis and V. Kumar. MeTiS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science, University Minnesota, 1995.

[17] P. Liewer and V. Decyk. A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes. *Journal of Computational Physics*, 2:302–322, 1985.

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

[19] J. Malone. Automated Mesh Decomposition and Concurrent Finite Element Analysis for Hypercube Multiprocessors Computers. *Eng.*, 70:27–58, 1988.

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

[21] S. Nolting. Nonlinear Adaptive Finite Element Systems on Distributed Memory Computers. *Proceedings of European Distributed Memory Computing Conference*, pages 283–293, April 1991.

[22] C. Ou and S. Ranka. Parallel Remapping Algorithms for Adaptive Problems. *Frontiers '95*, pages 367–374, 1995.

[23] C. Ou, S. Ranka, and G. Fox. Fast Mapping And Remapping Algorithm For Irregular and Adaptive Problems. *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, pages 279–283, Taipei, Taiwan, December 1993.

[24] A. Pothen, H. Simon, and K. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal of Matrix Analysis and Application*, 11:430–352, July 1990.

[25] J. Salmon. Parallel Hierarchical N-Body Method. Technical report, Center for Research in Parallel Computing, Caltech, Pasadena, CA, 1990.

[26] J. Salmon and D. Warren. personal communication.

[27] R. Shankar and S. Ranka. Hypercube algorithms for quadtree operations. *Journal of Pattern Recognition*, pages 741–747, September 1992.

[28] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[29] H. Simon and C. Farhat. TOP/DOMEC; A Software Tool for Mesh Partitioning and Parallel Processing. Technical report, NASA Ames Research Center, 1993.

[30] C. Thompson and H. Kung. Sorting on a mesh-connected parallel computer. *comm. ACM*, 20:263–271, 1977.

[31] D. Walker. Characterizing the Parallel Performance of a Large-Scale, Particle-in-Cell Plasma Simulation Code. *Concurrency: Practice and Experience*, 1990.

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