Data Partitioning For Parallel Spatial Join Processing

Xiaofang Zhou, David J. Abel and David Truffet
CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, ACT 2601, Australia
{xiaofang.zhou, dave.abel, david.truffet}@cmis.csiro.au

Abstract

The cost of spatial join processing can be very high because of the large sizes of spatial objects and the computation-intensive spatial operations. While parallel processing seems a natural solution to this problem, it is not clear how spatial data can be partitioned for this purpose. Various spatial data partitioning methods are examined in this paper. A framework combining the data-partitioning techniques used by most parallel join algorithms in relational databases and the filter-and-refine strategy for spatial operation processing is proposed for parallel spatial join processing. Object duplication caused by multi-assignment in spatial data partitioning can result in extra CPU cost as well as extra communication cost. We find that the key to overcome this problem is to preserve spatial locality in task decomposition. We show in this paper that a near-optimal speedup can be achieved for parallel spatial join processing using our new algorithms.

Keywords: spatial join, parallel processing, data partitioning.

1 Introduction

Two of the most successful database research areas in the last decade are the parallel databases which can improve database performance drastically via parallel processing [6, 19], and the spatial databases which can support spatial data types such as points, lines and polygons[9]. Most major database vendors have already provided parallel processing services for their relational database products, and more and more database vendors are starting to integrate spatial data types into their mainstream products. The combination of these two trends defines the new and challenging research area of parallel spatial databases.

Spatial joins, which combine two spatial data sets by their spatial relationship such as intersection and enclosure, are among the most important operations in spatial databases. Their processing costs are intrinsically high because of three factors. First, spatial data are often large in size and therefore expensive to transfer from disk. Second, the spatial operations, often implemented using computational geometry algorithms, are CPU intensive. Third, most join algorithms efficient for one-dimensional data, such as the sort-merge and the hash join algorithms, cannot be applied to multi-dimensional spatial data. Efficient spatial join algorithms have long been the focus of spatial database research[4, 8, 17, 18, 20, 21]. However, parallel spatial join processing has not been studied until very recently[5, 11].

In this paper we consider parallel processing of spatial joins with particular attention to examining whether the techniques which are efficient in parallel relational databases, such as the data partitioning methods and load-balancing algorithms, can be applied to spatial joins. Data partitioning parallelism has been successfully explored in parallel relational databases[6]. The basic idea to join relations R and S (denoted as $R \bowtie S$) in parallel is to partition R and S, by hashing or sorting on their join columns, into buckets $R_1, R_2, \ldots R_n$ and $S_1, S_2, \ldots S_n$ such that for all $1 \leq i, j \leq n$, $R_i \cap R_j = \emptyset$ and $S_i \cap S_j = \emptyset$ if $i \neq j$, and $R \bowtie S \equiv \cup_{i=1}^{n}(R_i \bowtie S_i)$. Such a data partitioning method has the *single assignment, single join* (SASJ) property as one tuple can be mapped into one and only one bucket, and one R bucket only needs to join with one S bucket. Each pair of R and S buckets, called a *task*, can then be assigned to a processor for parallel execution[16]. For joining spatial data which have extents, unfortunately, there does not exist a data-independent SASJ partitioning method. This can be seen by considering an example where two polygons A and B are mapped into different buckets using an SASJ method and then a polygon C, which overlaps with both A and B, is added into the data set. Therefore, a spatial data partitioning method is either *multi-assignment, single-join* (MASJ), or *single-assignment, multi-join* (SAMJ)[18]. A typical partitioning method for spatial data is based on *space decomposition*. That is, the space is decomposed into regions such that spatial objects can be partitioned according to their spatial relationship, such as enclosure and overlapping, with these regions. The R-tree approach ([10]) is SAMJ in the sense that an object is only assigned to one R-tree node but in order to find objects in a given area it is possibly necessary to search more than one R-tree node[4]. On the other hand, the R^+-tree approach ([23]) and the Z value based approach ([3, 20]) can be regarded as MASJ. The former keeps regions of tree nodes non-overlapping but assigns an object to all the nodes whose regions overlap with the object; and the latter can associate an object with multiple regions which are possibly at different resolution levels. While both MASJ and SAMJ data partitioning methods are popular as spatial indexing mechanisms[9], we argue in this paper that the MASJ type of data partitioning method is preferable for the purpose of parallel spatial join processing. In comparison with the task decomposition methods for one dimensional data, the multi-assignment spatial data partitioning methods have a distinct property - *object duplication*. That is, there can be duplicated objects among both the operand buckets and the results of different tasks (i.e., processors). Object duplication can impact negatively on the performance of parallel spatial join algorithms by the extra communication cost for sending one object to many processors and by the extra workload for processing the same pair of objects at different processors. Therefore, a data partitioning algorithm for parallel spatial join processing should not only balance the workload among tasks as most parallel relational join algorithms do, but also minimise the total workload which can vary depending on the partitioning methods.

Binary polygon intersection joins are considered in this paper, though our results can be extended easily to many other types of spatial joins. We concentrate on some fundamental issues which are general to data partitioning parallel spatial join algorithms instead of focusing on a specific parallel architecture. Our aim is to maximise CPU parallelism and minimise communication cost. While most previous parallel spatial join algorithms have largely evolved from sequential spatial join algorithms with the focus on utilisation of certain types of spatial indices, we adopt a different approach based on simple space partitioning. The load-balancing issue with non-uniform data distribution in the join columns, known as the *data skew* problem [15], is approached through a low-cost bottom-up merging process instead of the top-down recursive decomposition methods used by previous research. Combining the data partitioning techniques widely adopted by parallel relational databases and the filter-and-refine strategy for spatial operation processing, we present a seven-step framework for parallel spatial join processing. We identify a set of optimisation objectives by examining each step of the framework. Several novel algorithms are proposed for each step to achieve these objectives. The central idea, not surprisingly, is to preserve spatial locality during task decomposition. Using real spatial data, a performance analysis of the parallel join algorithm based on our framework shows a near-optimal speedup for CPU cost and only a moderate increase of total amount of data exchanged among processors when the number of processors increases. Our main contribution in this paper is to show that the parallel join algorithms which have proven successful for relational databases can also be applied to spatial joins with certain modifications. Both the simulation results and the results of running the algorithms on Fujitsu AP-1000, an experimental MIMD (Multi Instruction Multi Data streams) parallel computer with 128 configurable processors [14] show that our parallel spatial join algorithm can achieve a near-optimal speedup provided that the communication bandwidth is sufficient.

In Section 2 we give a brief review of related work. A framework for parallel spatial join algorithms is given in Section 3, together with a set of optimisation objectives regarding to the CPU costs and communication costs. Various design alternatives for each step of the framework are discussed in Section 4. In Section 5, the performance of a complete parallel spatial join algorithm based on the framework using the algorithms identified as most efficient for each step is analysed with real spatial data. The performance results of running our algorithms on Fujitsu AP-1000 are also reported in this Section. Our conclusions are presented in Section 6.

2 Related Work

2.1 The Filter-And-Refine Strategy

A common strategy to reduce the CPU and I/O costs for spatial join processing is the *filter-and-refine* approach which employs a filtering step followed by a refinement step. In the filtering step, a weaker condition for the spatial operation is applied to approximations of spatial objects to produce a list of *candidates* which is a superset of the final join results. These candidates are then checked by applying the spatial operation on the full descriptions of the spatial objects to eliminate the "false drops". The join cost can be reduced because the weaker condition is usually computationally less expensive to evaluate and the approximations are smaller in size than the full geometry of spatial objects. A common weaker condition for many spatial operations is the intersection of *minimum bounding rectangles* (MBRs) of spatial objects. Although the MBR intersection operation can be implemented as a pure aspatial operation using the coordinates of MBRs, the filtering cost can still be very high because those efficient relational join algorithms cannot be applied to the MBR intersection joins. Spatial indices can be used to reduce the filtering cost. It is proposed to sort the objects by their Z values and to merge them sequentially just like in the conventional sort-merge join algorithm except that a controlled traversal is required for the objects whose Z values are represented at higher than the resolution level[20]. Brinkhoff, Kriegel and Seeger propose to use R-trees for the filtering operation[4]. They identify overlapping MBRs at two levels using the plane-sweep algorithm. First, the MBRs for the R-tree leave nodes are examined to identify all R-tree node pairs, each from one operand table, with overlapping regions. Then, the objects within a pair of R-tree nodes are joined to produce candidates using the similar algorithm. The idea to organise spatial objects into a tree structure for efficient spatial join processing is generalised by Günther[8]. Lo and Ravishankar give an algorithm to build spatial indices to perform spatial join operations without spatial indices[17]. As the join operations are usually postponed to after selection operations by the query optimiser in a database system, the operand tables of a spatial join operation usually do not have spatial indices because they can be interim results of other operations. Some recent studies find that the data partitioning based spatial join algorithms using techniques similar to those used for relational data partitioning outperform the spatial join algorithms which build spatial indices on-the-fly [18, 21].

Previous research on spatial join algorithms has largely focused on the filtering step only. Partly because of this and partly because of the intrinsically high costs of the operations involved, the refinement cost becomes dominant for the queries with high join selectivity factors. Using the same weak condition, all filtering algorithms will produce the same set of candidates. However, different filtering algorithms may or may not produce duplicated candidates; and they may produce candidates in different orders. Such differences can influence significantly on the refinement cost. This problem has been recently studied by Abel *et al* [1].

2.2 Parallel Join Processing

A query is usually executed as a sequence of operations (e.g., fetch, sort and merge). While different queries and independent operations within a query can be executed in parallel, some consecutive operations in a query can also be executed concurrently by overlapping the execution (i.e., *pipe-lining parallelism*). Additional parallelism, known as *data partitioning parallelism*, can be obtained for set-oriented operations by partitioning the input data and applying the operator on part of data. Data partitioning parallelism is well positioned to explore the benefits promised by modern parallel computers [6]. Assuming the data are distributed among processors before join processing starts, a typical data-partitioning parallel join algorithm for relational data has three phases:

1. *the task decomposition phase*: the join operand data are partitioned in parallel into buckets, and bucket pairs are grouped into tasks. It is critical to maintain load-balancing among tasks such that all processors can finish at the same time in the third phase.
2. *the data redistribution phase*: the operand data are redistributed among processors such that the processors can execute their tasks in phase 3 independently.
3. *the parallel task execution phase*: the processors execute the tasks assigned to them in parallel. No communication is required unless phase 1 fails to balance the workload among processors. In that case, workload need to be re-balanced dynamically during execution.

Many task decomposition algorithms are proposed for relational joins (see [25] for a survey). When the data distribution expected by the data partitioning method differs from that in the join operand data, some tasks may contain more data than others and thus take longer time to process[15]. A data partitioning method is said to be *balanced* if it does not result in fluctuation of workload among tasks even when data skew is present. The algorithms to handle data skew can be classified into two categories: those using sampling techniques to find a tailor-made data partitioning method for a given join operation [7], and those by decomposing data into much smaller units and merging them into larger but balanced tasks [13, 15]. Dynamic load-balancing inevitably interrupts parallel task execution. Its cost can be high because of dynamic workload monitoring and additional work for data redistribution. It is not used unless the join workload is unpredictable due to a lack of sufficient statistics data about the operation.

Spatial data skew is not uncommon. For instances, most land parcels and utility lines are around large cities. Therefore, it is important for a spatial data partitioning method to be data skew robust. In order to prevent load imbalance, two approaches can be used for partitioning spatial data:

1. *the top-down approach*: the space is partitioned into a number of regions, each corresponding to a bucket. Once a bucket is found to have more than expected workload (e.g., in terms of number of objects), the region for the bucket is split and the objects inside are re-assigned to the new buckets according to the new region definition. Some rules are required to reduce the possibility of further splitting, or to ensure the smaller regions contain the same amount of objects. In order to end up with a required number of buckets, an algorithm to merge buckets after splitting may be necessary. Most spatial index mechanisms, such as the R-tree and R$^+$-tree, decompose the space in the top-down manner recursively. Hoel and Samet adopt this approach to balance amount of spatial data by building a PMR-quadtree using recursive object decomposition [11]. The process of building the PMR-quadtree is parallelised by using a special parallel platform.
2. *the bottom-up approach*: the space is initially decomposed into N cells where $N >> n$, n is the number of processors. After the objects are assigned into N small buckets, these small buckets are merged into n larger but balanced buckets (usually applying heuristic rules to the *bin-packing* problem [12]). This approach is widely used to handle the data skew problem in parallel relational join algorithms [13, 15]. It is also used by Patel and DeWitt to partition spatial data for sequential join processing [21].

We consider the bottom-up approach in this paper because it is simple and each object is assigned to one or more buckets once only. In addition to finding a proper spatial data partitioning method, other new issues related to the design of a load-balanced bottom-up task decomposition algorithm for parallel spatial join processing include estimating spatial join workload and handling object duplication.

The algorithm in [5] is also based on data partitioning. However, it is completely different from our work on three grounds: 1) their task decomposition is based on pre-existing R-tree indices while we do not assume any type of spatial indices on the join columns; 2) they solve the load-imbalancing problem by rebalancing workload dynamically using global virtual memory while we prevent the data skew problem in the task decomposition stage; and 3) they do not consider refinement operation (by assuming the candidates are refined at where they are produced) while we consider the complete process of spatial join processing. Our work is similar to [11] in two aspects: both explore CPU parallelism in data parallel environments and neither relies on pre-existing spatial indices. Their work uses top-down object decomposition to balance the join workload. This requires constant data exchange during the partitioning stage, and so requires a specialised parallel platform. Our work, however, is based on bottom-up data partitioning and does not decompose spatial objects. Our algorithms do not rely on specific parallel architectures.

3 Parallel Spatial Join Algorithms

3.1 Assumptions

Let the scheme of the operand relations be

$$(id : \text{number}, mbr : \text{rectangle}, size : \text{number}, boundary : \text{polygon})$$

where id is the identifier of the spatial object; mbr is represented by the lower left corner (x_{low}, y_{low}) and the upper right corner (x_{high}, y_{high}) of the MBR of polygon; and $boundary$, represented as a sequence of vertices, describes the geometry of the polygon. The number of vertices, $size$, is introduced for the purpose of workload estimation. Columns (*id*, *mbr*, *size*) are called the *key-pointer* data. The join we consider is of the form $\pi_{R.id,S.id}(R \bowtie_{R.boundary\ intersect\ S.boundary} S)$.

We assume a shared-nothing architecture in this paper[6]. It consists of a set of n identical parallel processors $p_1, \ldots p_n$ interconnected by a communication network, through which and only through which different processors can exchange data. We also assume that the cost of sending the same amount of data between any two processors are identical. Each processor has its own memory and can execute tasks independently (i.e., MIMD parallelism). As in [4], there is a processor p_0 designated as the interface processor which can communicate with every other processor directly and simultaneously. The parallel computer is connected to the outside by p_0 (i.e., to receive data from the database server where spatial data are stored). We assume the initial data distribution among the parallel processors is *even*. That is, before the parallel join processing starts we have $R = \cup_{i=1}^{n} R_i$, $S = \cup_{i=1}^{n} S_i$, R_i, S_i reside on p_i, $1 \leq i \leq n$, and $|R_1| = \ldots = |R_n|$ and $|S_1| = \ldots = |S_n|$ where $|R|$ denotes the number of objects in R. Further, we assume all the data can be stored in the aggregate memory of the parallel processors. Although it is known that the join cost can be reduced by distributing spatial data to processors in clusters (e.g., by locality), this is not always possible as the join operand tables are often the intermediate results from other database operations. Instead of considering the operation of data clustering (which can be time-consuming) as a pre-processing step, we consider it as part of the parallel join processing by assuming the initial distribution is *random* (i.e., the spatial objects in R and S are distributed into R_i and S_i, $1 \leq i \leq n$, in a round-robin way). Our assumptions about initial data distribution model the cases where the join operand data are the results of some previous operations, or where data can be bulk loaded into parallel processors from outside (e.g., a parallel file system or a database server). Final join result collection is not considered. These assumptions are common for parallel join research[11, 13, 15].

3.2 A Framework

The filter-and-refine strategy in multiprocessor can, in addition to reducing CPU and I/O costs, reduce the amount of data to be transferred among processors (this has been demonstrated in a distributed spatial database environment by Abel *et al* [2]). It is natural for a parallel spatial join algorithm to adopt this two-step approach. In light of the paradigm of parallel relational join algorithm and the filter-and-refine spatial operation processing strategy, we propose a framework for parallel spatial join processing. The framework consists of a *parallel filtering phase* (steps F1 - F4) and a *parallel refinement phase* (steps F5 - F7). Only the key-pointer data are used in the filtering phase. The candidates in the refinement phase determine which spatial objects are required for this phase and where they are to be used. Each phase has three steps (i.e., task decomposition, data redistribution and task execution). The filtering phase has an additional step to redistribute its results (i.e., candidates). The seven-step framework for n processors is as follows:

1. **F1** (*filtering task decomposition*): each p_i , $1 \leq i \leq n$, partitions the key-pointer data of R_i and S_i into R_{ij}, S_{ij}, where $1 \leq j \leq N$ and $N >> n$, using a **spatial data partitioning** algorithm which defines a mapping between objects and regions (called *cells*). Then, p_0 collects the sizes of small buckets R'_i and S'_i, $1 \leq i \leq N$, where $R'_i = \cup_{j=1}^{n} R_{ji}$ and $S'_i = \cup_{j=1}^{n} S_{ji}$, and merges the small buckets into n large buckets R''_i and S''_i, $1 \leq i \leq n$, using a **bucket merger** algorithm. We denote the indices of the large R and S buckets which the i^{th} small R and S bucket is merged into as $f(i)$ and $g(i)$ respectively (unlike in parallel relational join algorithms, one small bucket can be merged into multiple large buckets here).

2. **F2** (*key-pointer data redistribution*): each p_i sends the key-pointer data for the objects in R_{ij} and S_{ij} to processors $p_{f(j)}$ and $p_{g(j)}$, $1 \leq j \leq N$, and puts the data received into R''_i and S''_i with the processors from which the data come from recorded.

3. **F3** (*parallel filtering*): each p_i performs its filtering task $R''_i \bowtie_{MBR\ intersect} S''_i$ to produce candidates $\{(r.id,\ s.id,\ cost,\ cell)\}$ using an **MBR intersection join** algorithm, where *cost* is the estimated refinement cost of the candidates (see Section 4.5), and *cell* is the cell number of object *r.id* (if there is more than one cell for the object, choose the smallest number). The duplicate candidates produced locally, if any, are removed at each processor independently.

4. **F4** (*candidates distribution*): each p_i sends the *full candidate* (*r.id*, *s.id*, *cost*, *cell*) to the processor which has object *r.id* (we call it the *home processor* of object *r.id*), and sends the *data-request candidate* (*s.id*, *cell*) to the home processor of object *s.id* if the home processors of *r.id* and *s.id* are different.

5. **F5** (*refinement task decomposition*): each p_i processes the full candidates it receives to remove the duplicate candidates produced by different processors (which cannot be removed in step F3). The refinement cost for a cell is calculated in parallel by adding up the costs of all full candidates a processor receives with the cell number. A cell-to-task mapping is determined at p_0 using a **workload re-balancing** algorithm which takes as input the estimated refinement workload of each cell.
6. **F6** (*full object redistribution*): each p_i sends full objects used by either full candidates or data-request candidates to their destination processors according to the cell-to-task mapping determined at F5. The full candidates (but not the data-request candidates) themselves are also sent to the destination processors.
7. **F7** (*parallel refinement*): each p_i performs its refinement task using the full candidates and the full objects received using a **polygon intersection check** algorithm.

In this framework, most duplicate candidates are removed at where they are produced before they are sent back to their home processors. The remaining duplicate candidates (produced by different processors) are removed at home processors. As all the candidates referencing to the same R object have the same home processor, there are no duplicate candidates after these two steps. Note that data-request candidates do not participate in duplicate removal or refinement cost estimation; they are only used for requesting S objects.

Clearly, the performance of a parallel spatial join algorithm based on the above framework depends on the algorithms it uses for each step. Before we discuss these algorithms in the next section, we analyse the costs of these steps to identify a set of optimisation objectives.

3.3 Optimisation Objectives

The complex problem of modelling the cost of a parallel spatial join algorithm can be simplified by assuming synchronised steps (i.e., no processor can proceed to the next step until all processors finish the current step). Therefore, the total cost is the sum of the costs for the component steps. Now we analyse the communication costs and the CPU costs for these steps.

Data exchange in step F1 occurs between p_0 and the parallel processors: the parallel processors send their numbers of local objects in each cell to p_0 and p_0 broadcasts back cell-to-filtering-task mapping information. Clearly, the communication cost of this step is solely determined by the number of cells. As each object is processed only once in step F1 because of the bottom-up data partitioning approach, the CPU cost of step F1 is simply the cost of a linear scan for local R and S objects to map them into small buckets (experiments show that the CPU cost difference between multi-assignment and single-assignment using MBRs is negligible). Bucket merging is done at p_0. Under the assumption of even initial distribution, the CPU cost for this step is uniform across all processors and independent of the partitioning algorithms used.

In step F2, the key-pointer data are re-organised before being transferred to other processors (according to the cell-to-filtering-task mapping), and the objects duplicated in different small buckets sending to the same destination are removed. The CPU costs for removing duplicates are uniform among processors because of even data distribution. However the number of data sent by a processor can vary depending on the cell-to-filtering-task mapping as one object can be sent to several processors if the object is assigned to multiple filtering tasks. The communication cost of this step depends directly on the bucket merger algorithm which defines the filtering tasks, and indirectly on the data partitioning algorithm that determines the input to the bucket merger algorithm. Under the assumption of even and random initial data distribution, each processor has an equal share of the extra amount of data exchanged, which is equivalent to the total number of objects duplicated in the filtering tasks. Therefore, we need, as the first optimisation objective, to *minimise object duplication among filtering tasks.*

Step F3 incurs no communication cost. The CPU cost is determined by the filtering task which requires the largest number of MBR comparisons. Thus the merger algorithm must *minimise the number of MBR comparisons for filtering tasks.* In step F4, candidates are sent back to home processors for requesting objects for refinement. The communication cost of this step, clearly, depends on the number of candidates duplicated among processors. Therefore, the merger algorithm must also *minimise the number of candidates duplicated among filtering tasks.* Another factor affecting the communication cost of this step is whether some processors

Num of points	1-10	10-30	30-50	50-100	100-500	500-1000	1000+
Num of polygons	18564	36158	11027	7873	5009	444	310

Table 1: Description of the Sequoia polygon data

produce significantly more candidates than others (this is known as the *join product skew* problem [25]). Prevention of this problem requires estimation of the MBR join selectivity for filtering tasks; this is beyond the scope of this paper.

The cost of step F5 is similar to that of step F1 except for an additional operation used for removing duplicate candidates. This cost can be regarded as uniform among processors because of even and random initial data distribution. Refinement task creation is a centralised operation at p_0, and data exchange occurs only between p_0 and the parallel processors. Again, the amount of data communication is determined only be the number of cells (i.e., sending the estimated refinement cost for each cell to p_0 and receiving candidates-to-refinement-task mapping from p_0). Full geometry descriptions of the spatial objects need to be transferred in step F6. A local processing is performed to determine which objects to send where and to remove the redundant objects sending to the same processor. Note that this cost is independent of the filtering algorithm after the duplicated candidates are removed. Under the assumption of even and random data distribution, the objective of minimising the communication cost of this step can be stated as to *minimise the number of objects duplicated among refinement tasks*, which means not sending one object to multiple processors. Step F7 incurs only computational costs. The optimisation objective for this step is obvious: *balancing the workload among refinement tasks.*

4 Design Issues

In this section we discuss the algorithms to be used for each step in the parallel join framework to achieve the optimisation goals identified in Section 3.

4.1 Data Sets

The algorithms discussed in this section are compared using the Sequoia data set[24], which are publicly available and have been used by several published spatial join performance analysis [4, 21]. It consists of two sets of polygons: the "land use" polygons (denoted as R hereafter) that describe land uses in California, and the "island" polygons (denoted as S) which are the holes in land use polygons (so all polygons are simple polygons). Data distribution of the data set is illustrated in Fig 1 and 2. There are 37 land use types with from a few to more than 20000 polygons in each type. The data set contains 79385 polygons ($|R| = 58411$, $|S| = 20974$) with 3.65 million vertices in total (see Table 1 for details). A vertex is represented as a pair of integers. The total amount of data, including the key-pointer data, is 29.67 MBytes. The ratio of the area of the spatial extent over the average MBR size is about 248×248, 282×282 for R and S data respectively. Each object is assigned with a unique identifier by concatenating the land use code and the sequence number of the object in the land use type. Note that the sequence numbers bear a co-relationship with spatial locality (i.e., for the objects of the same land use type, the objects with "close" sequence numbers are close in the space). The query used in this paper is, as in[21], to find all the holes in polygons. The MBR filtering phase returns 84862 distinct candidates, which reference to 54015 different objects; and the final result returns 47533 pairs of polygons.

4.2 Local Join Algorithms

As does any filter-and-refine based spatial join algorithm, our framework uses two join algorithms: an MBR intersection join algorithm to find candidates and a polygon intersection join algorithm for refinement. The MBR intersection join algorithms have been extensively investigated in the literature. The algorithms based on spatial indices are not attractive to our parallel spatial join framework for the reasons discussed in Section 1.

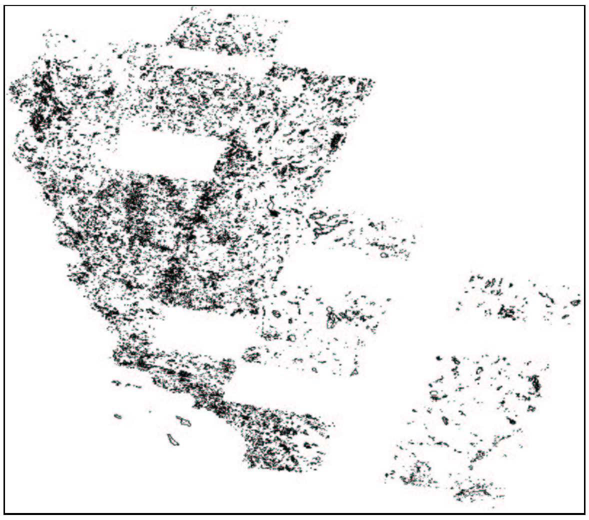
Figure 1: Sequoia island polygons.

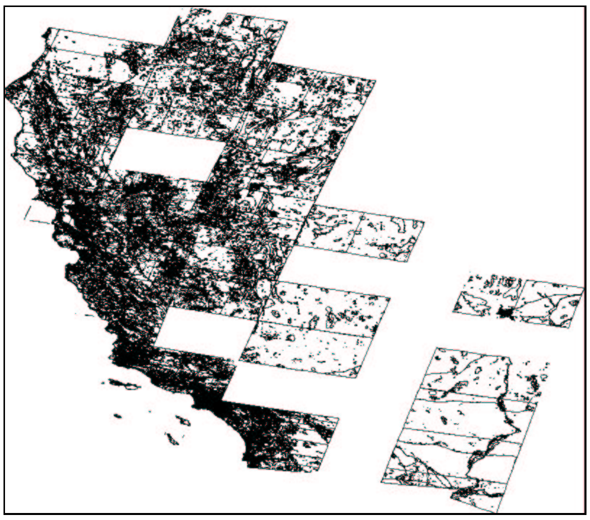
Figure 2: Sequoia land use polygons.

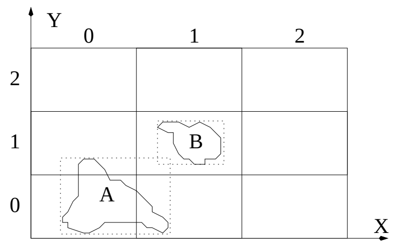

Figure 3: The space is divided into 3 × 3 cells.

Because a large bucket is merged from a set of small buckets, it is natural for us to use the partitioning-based MBR join algorithm [18, 21]. That is, a pair of large buckets is joined by joining each pair of the component small buckets which correspond to the same cell; and a pair of small buckets are joined using the nested-loops method (i.e., to check each object in one small bucket with all objects in another bucket) because the sizes of small buckets are small enough to yield satisfactory performance.

The polygon intersection algorithm as a computational geometry problem has been well studied. It is a consensus that the most efficient algorithm is the plane-sweep algorithm, which requires $O((m+n)\log(m+n))$ time to check polygons with m and n vertices respectively[22]. As we assume that all the objects can be held in the distributed memory of parallel processors, candidate sequencing ([1]) is not considered here. In other words, the polygon intersection join algorithm used in this paper is to process sequentially the candidates assigned to a processor; and for each pair of polygons, the plane sweep algorithm is used to check if the polygons have cutting lines after whether a vertex of one polygon is inside another are checked.

4.3 Spatial Data Partitioning

As mentioned before, a spatial data partitioning method can be either MASJ or SAMJ. Partitioning the *spatial extent* of the join operand data (i.e., the MBR for all the polygons in both tables) into rectangles (i.e., *cells*) of the same size (see Fig 3), an MASJ partitioning method maps all objects overlapping with a cell into the bucket for the cell; and an SAMJ method groups objects according their geometric centers. For example, object A in Fig 3 is mapped to cell 00 using the SAMJ method, and to cells 00, 01, 10, 11 under the MASJ mapping. Object B is mapped into cell 11 for both methods. A spatial data partitioning method can affect the number of MBR comparisons required for the MBR intersection join, the number of duplicated objects among large buckets and the number of duplicated candidates produced by filtering tasks. We compare these two partitioning methods by these three factors.

4.3.1 Number of MBR Comparisons

Fig 4 illustrates how the number of MBR comparisons can be reduced using different data partitioning methods (not that the size of each bucket is typically different from the expected size due to data skew, and these sizes vary under different partitioning methods). The workload of $R \bowtie_{MBR\ intersect} S$ is represented by the shaded area in Fig 4. The outer unshaded rectangle is the MBR join workload when the nested-loops algorithm is used. Obviously, an SASJ method does not result in overlapping objects between different buckets, thus no redundant join workload presents under that method. This method is, however, not applicable for spatial joins as we mentioned before. Both MASJ and SAMJ methods incur some redundant join workload. In Fig 4 (b), bucket R_2 needs to join with both S_2 and S_3 (i.e., multi-join). In Fig 4 (c), there can have duplicated objects between neighbouring buckets (i.e., the same pair of objects can be duplicated in several filtering tasks).

Now we show that the SAMJ method usually requires a larger number of MBR comparisons than the MASJ method. First, we consider the simple case where both operand tables have uniform data distribution. Let the space be partitioned into n cells. For the SAMJ method, we have $R = \cup_{i=1}^{n} R_i$ and $S = \cup_{i=1}^{n} S_i$, and for any $1 \le i, j \le n$, $R_i \cap R_j = \emptyset$ and $S_i \cap S_j = \emptyset$; for the MASJ method, we have $R = \cup_{i=1}^{n} R'_i$ and $S = \cup_{i=1}^{n} S'_i$ where for some $1 \le i, j \le n$, $i \ne j$, $R'_i \cap R'_j \ne \emptyset$ and $S'_i \cap S'_j \ne \emptyset$. Further, we assume all bucket sizes are about the same after partitioning. Combining these assumptions, we have $|R_i| = |R|/n$, $|S_i| = |S|/n$, $|R'_i| = (1+\alpha)|R|/n$, and $|S'_i| = (1+\beta)|S|/n$, where α and β are the percentage of object duplication caused by multi-assignment in R and S buckets respectively. Therefore, the total number of MBR comparisons for MASJ and SAMJ mapping methods can be expressed as

$$C_{MASJ} = \sum_{i=1}^{n} |R'_i||S'_i| = \frac{(1+\alpha)(1+\beta)}{n}|R||S|, \qquad C_{SAMJ} = \sum_{i=1}^{} n|R_i||S_i| = \frac{k}{n}|R||S|$$

where k is the average number of S buckets for a R bucket to join with under the SAMJ mapping. Thus, $C_{MASJ} \le C_{SAMJ}$ if $(1+\alpha)(1+\beta) \le k$. When $\beta = 0$ (i.e., object duplication occurs only in one table), $C_{MASJ} \le C_{SAMJ}$ if $\alpha \le k-1$; and when $\alpha = \beta$ (i.e., object duplication occurs for both tables and the two tables have similar data distribution in the space), $\alpha \le \sqrt{k} - 1$. Under the SAMJ mapping, one R bucket can join with as many S buckets as its extent intersects with. Assume $k = 3$, then the MASJ method has a smaller search space if the average percentage of overlapping objects among buckets are less than 200% and 73% for the above two cases. Moreover, when n is large, another join algorithm (e.g., the plane-sweep algorithm in[4]) is needed to find which pair of buckets need to be joined when the SAMJ method is used.

The total number of MBR comparisons required by the SAMJ and MASJ partitioning methods for the Sequoia data is shown in Fig 5. The data partitioning costs for the two methods are practically identical. When n increases from 100×100 to 400×400, k is reduced from 4.74 to 3.90 for the SAMJ methods, and the total comparison numbers for the two methods are getting closer. However, the average number of objects in each bucket reduces to 1 when $n > 300 \times 300$, and the bucket management cost (e.g., to determine which buckets are to be joined) becomes dominant. When the number of cells is not too large (e.g., no more than 200×200 cells for the Sequoia data), the benefit of the MASJ method is clear.

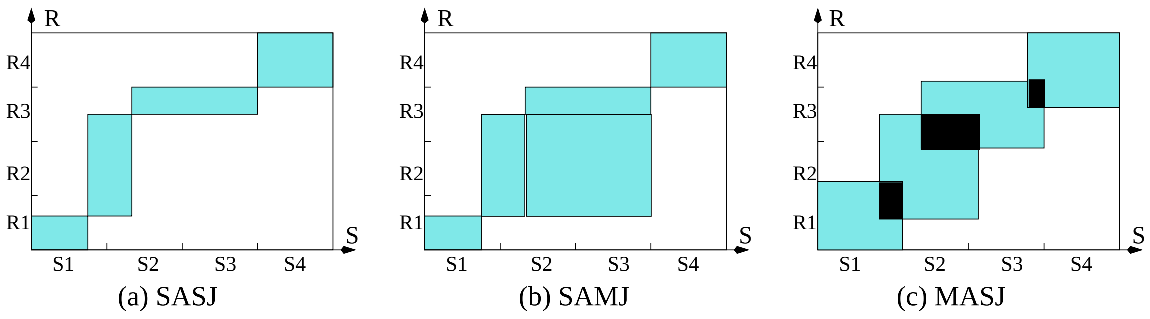

Figure 4: Workload reduction by data partitioning.

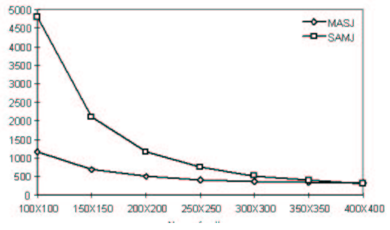

Figure 5: Number of MBR comparisons (×1000) against the number of cells.

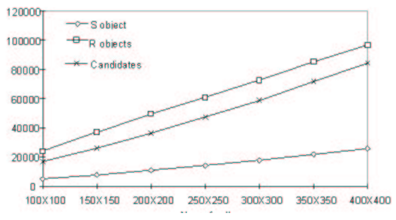

Figure 6: Number of duplicated objects and candidates against the number of cells.

4.3.2 Object Duplication

The benefit of not having duplicated objects in small buckets for an SAMJ method does not imply that there is no object duplication among the filtering tasks as a small bucket from one table often needs to be duplicated among several filtering tasks because of multi-join. In addition to a higher CPU cost as seen above, this also means a higher communication cost. For example, if there are some objects overlap with both region i and j, all R_i objects (including those not overlapping with region j) need to be sent to where S_j resides unless these objects can be distinguished from each other (e.g., using the algorithm in[4]). This will require additional computation and communication costs. An SAMJ method can have a particularly poor performance when there are some large objects spanning over many cells.

Fig 6 shows the trend of object duplication against numbers of cells in the filtering step under the cell-overlapping MASJ mapping. While the objects duplicated in both operand tables and in the resultant candidate set rise steadily with the increase of cell numbers, the number of MBR comparisons does not drop at the same rate (see Fig 5). Recalling the ratio of the spatial extent over object sizes given in section 4.1, it is clear that the cell size should not be smaller than the average size of MBRs.

One advantage of an SAMJ method is that it does not produce duplicate candidates across processors. However we found that the cost of removing duplicate candidates negligible in comparison to the cost caused by a significantly larger search space resulted from an SAMJ method as well as caused by bucket management. An MASJ method is clearly the preferred data partitioning method for parallel spatial join processing. We use the cell-overlapping MASJ partitioning method in the rest of this paper. Next we show that object duplication associate with the MASJ method can be greatly reduced by using a proper bucket merger algorithm.

4.4 Merging Buckets

The bucket merger algorithm merges N small buckets into n filtering tasks to balance the workload and minimise duplication of the key-pointer data among filtering tasks. Note that the problem of merging small buckets into balanced large buckets alone is an NP-hard problem. We consider three heuristic-based algorithms here:

1. *the round-robin algorithm* (RR): use the round-robin method to merge small buckets into large buckets. This method, originally proposed to handle data skew in parallel relational join processing by Kitsuregawa[16, 15], is used by Patel and DeWitt for spatial data partitioning [21].
2. *the Z Order greedy algorithm* (ZOrder): associate each small bucket with the Z value of the corresponding cells, and sort the small buckets by their Z values. Then, as many as consecutive small buckets (in the sorted order) are merged into a large bucket until a pre-set large bucket capacity C is reached. If there are any small buckets left after n large buckets are formed, the remaining small buckets are assigned, one by one, into the large bucket with the smallest estimated load. This algorithm minimises object duplication by merging together neighbouring cells (which are close to each other after sorting by the Z values), and balances the workload using the greedy heuristic. We set $C = \sum_{i=1}^{n} |R_i|/n$ and increase the workload of the i^{th} large bucket C_i after small bucket R_j is merged into it using formula $C_i = C_i + |R_j|$.
3. *the improved Z Order greedy algorithm* (ImpZOrder): the ZOrder algorithm does not consider object duplication in large bucket size estimate. In order to improve the accuracy of large bucket size estimation,

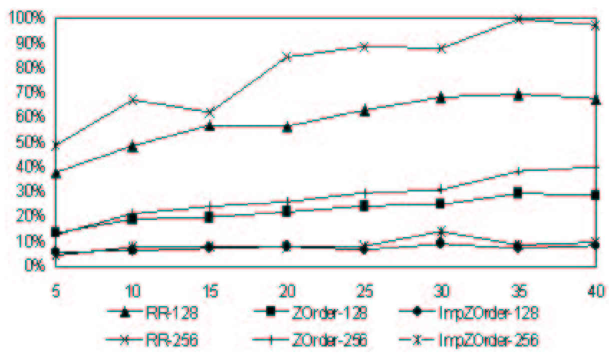

Figure 7: Variation of large bucket sizes against number of filtering tasks.

the number of duplicated objects needs to be recorded in the data partitioning algorithm by recording the number of objects overlapping between neighbouring cells. Let s_{ij} and e_{ij} be the numbers of duplicated objects between bucket R_{ij} and its southern and eastern neighbours respectively (note that when an object overlaps with both the southern and eastern cells, we record this duplication only once to avoid over-estimate the number of duplicate objects). We set $C = (1 + \gamma)|R|/n$ where γ is an allowance for duplicate objects in the merged buckets (we set $\gamma = 0.02$ in our tests), and $C_i = C_i + |R_j| - s_j - e_j$ after R_j is merged in.

The RR method is used to test if this commonly used method for parallel relational join processing is suitable for parallel spatial joins. The other two are used to test the influences of spatial locality and object duplication respectively. We ran two sets of experiments using the land use polygons (i.e., R) for $N = 128 \times 128$ and 256×256. In Fig 7, the bucket size *variation* is defined as $(B_{max} - B_{opt})/B_{opt}$ where B_{max} is the number of objects in the largest merged bucket, and B_{opt} is the optimal size which is the number of total distinct polygons in the data set divided by the number of buckets (i.e., $|R|/n$). Fig 7 demonstrates clearly that algorithm RR is not suitable for spatial data partitioning. It can increase bucket sizes by at least 40% because of object duplication. This overhead increases with the number of cells as well as with the number of filtering tasks. Algorithm ZOrder can reduce this overhead significantly. However, it fails to balance the load among buckets because of inaccurate estimation of object numbers after merging. By considering the number of objects duplicated among small buckets, algorithm ImpZOrder can reduce size variation to less than 10%. Note that this method is also robust to the number of buckets and the number of cells. Fig 8 gives the actual number of distinct objects in the largest merged bucket merged from 256×256 cells.

4.5 Re-Balancing Workload for Refinement

A refinement task is defined as a set of candidates. Two criteria for the refinement task creation step are to minimise the communication cost by minimising the number of objects duplicated among refinement tasks and to minimise the response time by balancing refinement workload. In this subsection we examine the impact of possible join product skew in the filtering phase before giving an algorithm to overcome this problem.

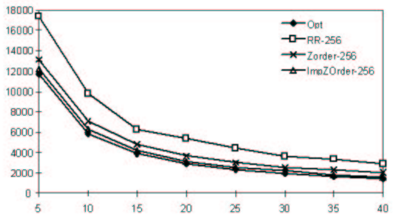

Figure 8: Maximum large bucket sizes of merging 256×256 cells against the number of filtering tasks.

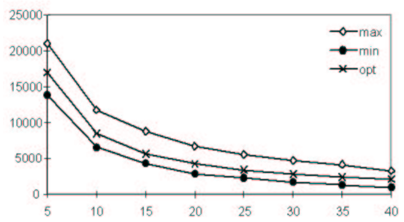

Figure 9: Numbers of candidates produced by varying number of filtering tasks.

4.5.1 MBR Join Selectivity

As we will further discuss later, the refinement workload is determined by not only the number of candidates but also the number of vertices of the objects referenced by the candidates. The spatial data partitioning algorithm and the bucket merger algorithm in the filtering phase cannot balance the refinement workload because the MBR join selectivity are unknown at that stage. MBR join selectivity estimation requires the statistical data about the operand tables such as the object sizes and spatial distribution of the objects. Even for the bucket merger algorithm which tries to balance the number of objects in the filtering tasks, Fig 9 shows that there still exists a significant amount of product skew caused by non-uniform MBR join selectivity in the filtering tasks, which are created using the MASJ partitioning method merged from 256×256 cells using algorithm ImpZOrder. One can see that some filtering tasks can produce as many as 50% more candidates than the optimal case (which is the total number of distinct candidates divided by the number of processors). Fig 9 also shows that the proportion of the difference between the maximum and minimum numbers of candidates among filtering tasks becomes larger with the increase of the number of filtering tasks. This suggests that an algorithm of processing candidates where they are produced cannot lead to load-balanced refinement.

4.5.2 Object Duplication in Refinement Tasks

Now we consider how to reduce object duplication among refinement tasks. For two candidates referencing to the same object, the object is duplicated if the candidates are assigned to different refinement tasks. Object duplication here will increase the communication cost for sending the full geometry of spatial objects. One straightforward method is to sort the candidates by their identifiers (say, of R then S) and assign the candidates sharing the same R object to the same refinement task. This method, however, cannot at the same time avoid duplicating objects from another table. In order to minimise duplicated objects for both tables, we apply the same idea as used for bucket merging. That is, we sort the candidates by the Z values of their cells and group candidates into refinement tasks in the sorted order using certain load-balancing criteria (to be discussed later). The reason of doing so is that the candidates referencing to the objects which are spatially close to each other are more likely to share common objects for both tables.

Fig 10 shows the empirical results about duplication of full objects in refinement tasks, which are defined by merging 256×256 cells with the objective of balancing the number of candidates in refinement tasks using various methods (i.e., sorting by R identifiers, by S identifiers and by the Z values). It is interesting to notice the effect of relationship between object identifiers and their locations. As mentioned in Section 4.1, there exists a co-relationship between the identifiers of the Sequoia objects of the same land use type and the spatial locations of the objects. Therefore, sorting by the S identifiers effectively groups objects spatially close to each other together. Note that the S objects are regarded as one type of land use - "island". This is not true for R objects which are from 37 types of different land uses. As Fig 10 illustrates, sorting by the Z values of the cells is slightly better than sorting by the S identifiers, while both of them are significantly better than sorting by R identifiers and both are close to the optimal values (which is the total number of distinct objects used by the candidates divided by the number of tasks). Thus, grouping candidates by the Z order can greatly eliminate object duplication among refinement tasks. This is particularly useful when object identifiers bear no relationship with spatial locations or such a relationship is unknown to the query optimiser. It should be pointed out that the R identifiers here are not completely random as they are still spatially clustered for the

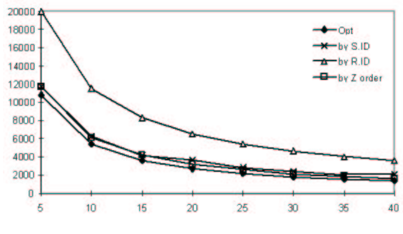

Figure 10: Maximum number of distinct objects against varying number of refinement tasks.

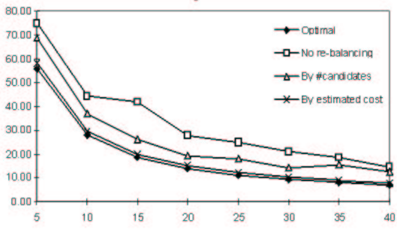

Figure 11: Response time (seconds) of parallel refinement using different re-balancing algorithms against the number of processors.

objects of the same land use (thus the results in Fig 10 can be better than the worst case with completely random identifiers).

In our refinement load-rebalancing algorithm, the candidates in a cell are grouped as a unit with the sum of costs of the member candidates as the unit cost. In comparison to considering each candidate as a unit for refinement task creation, our method can reduce the communication cost (from the parallel processors to p_0) as well as the task scheduling cost at p_0. It makes these costs dependent only on the number of cells, not the number of candidates.

4.5.3 Refinement Workload Estimation

As one can see from Table 1, the number of vertices in a polygon ranges from a few to thousands. Thus, it is inadequate to regard the refinement costs for different candidates as the same. It is difficult to give an accurate cost model for polygon intersection as the cost depends on whether two polygons intersect or not as well as how they intersect. Let m and n be the number of vertices in polygons r and s. For the best case the polygon join algorithm described in Section 4.2 takes only $O(n)$ time when the point of polygon r used for testing point-in-polygon is inside polygon s; and at the worst takes $O(m) + O(n) + O((m+n)\log(m+n))$ when the two polygons do not intersect. Here we use $m+n$ as the formula to estimate the polygon intersection cost. Fig 11 shows the refinement response time for the following three methods (the optimal response time is calculated by dividing the single-processor refinement time by the number of processors):

1. *no re-balancing*: refines candidates at where they are produced. Both local and inter-processor duplicate candidates are removed.
2. *re-balancing by candidate numbers*: the candidates are sorted by their Z values and grouped by that order into refinement tasks such that each task has the same number of candidates.
3. *re-balancing by estimated cost*: this method is similar to the above but balances the workload using the cost estimate function $m + n$. Costs are estimated when candidates are produced, and the cost for the candidates in a cell is calculated in parallel in step F5.

As shown in Fig 9, different processors can produce significantly different numbers of candidates. When the refinement workload is not re-balanced, as Fig 11 shows, the processors with larger numbers of candidates jeopardise the response time for parallel refinement. As the cost to check polygon intersection depends on the number of vertices in the polygons, refinement load-balancing cannot be achieved by only balancing the number of candidates among processors. Fig 11 shows that our cost function can reasonably reflect the real refinement cost, and a near-optimal speedup can be achieved for parallel refinement after workload re-balancing.

5 Performance

We have discussed the algorithms required by the parallel spatial join framework individually. In this section we examine the overall performance of the complete parallel spatial join algorithm based on the framework using the most efficient algorithm for each step (i.e., the cell-overlapping based MASJ algorithm with 128×128 cells for data partitioning, algorithm ImpZOrder for bucket merging and the Z order based greedy algorithm with $(m+n)$ as cost estimate for refinement workload re-balancing). After discussing the performance results obtained from simulation, we report the performance of the algorithm on Fujitsu AP-1000, a 128-processor MIMD parallel computer.

5.1 CPU Costs

First we look at the CPU costs for the operations involved in the parallel spatial join algorithm. We summarise these operations here. Each processor first partitions the key-pointer data of its local objects into buckets. The bucket sizes, together with the information about object duplication between neighbouring cells are sent to p_0 which merges the cells into filtering tasks. Each processor then reorganises the local key-pointer data according to the filtering task definition such that no redundant data are sent to the same processor. After the parallel MBR join processing using the cell-based nested-loops algorithm, each processor removes local duplicate candidates before sending them to the home processors to request full objects. The inter-processor duplicate candidates are removed after each processor receives all candidates from others. Candidates are grouped by cells in parallel, and mapped into refinement tasks at p_0. Finally, the full descriptions of the objects required by the candidates are packed at each processor (with duplicate objects for the same processor removed) and are sent out for parallel refinement.

Num of procs	Partition		Create filtering tasks	Reorganise key-pointer data		MBR join		Remove duplicate candidates		Create refinement tasks	Pack full objects		Refine	
	Total	Max		Total	Max	Total	Max	Total	Max		Total	Max	Total	Max
5	1.00	0.23	0.09	2.04	0.42	5.75	1.29	3.10	0.63	0.02	4.20	0.93	276.7	61.56
10	1.21	0.16	0.09	2.09	0.22	6.17	0.79	3.03	0.32	0.02	4.10	0.42	277.3	31.56
15	1.49	0.14	0.10	2.39	0.18	6.39	0.58	2.89	0.21	0.02	3.97	0.31	277.3	21.40
20	1.92	0.15	0.12	2.56	0.16	6.68	0.52	2.89	0.15	0.02	3.87	0.21	277.6	16.24
25	1.99	0.14	0.12	2.78	0.13	6.87	0.41	2.86	0.15	0.02	3.67	0.22	277.8	13.23
30	1.89	0.11	0.13	2.97	0.13	6.79	0.32	2.82	0.11	0.02	3.39	0.13	277.9	10.99
35	2.00	0.12	0.14	3.13	0.11	6.94	0.30	2.78	0.10	0.02	3.20	0.11	278.3	9.44
40	1.87	0.11	0.16	3.31	0.10	6.92	0.25	2.79	0.09	0.02	3.06	0.09	278.5	8.50

Table 2: CPU costs (seconds) for operations in the parallel spatial join algorithm.

Table 2 reports the total time and the time required by the slowest processor to perform the operations on the Sequoia polygons. These figures are obtained by running the algorithms on a SunSPARC 10 workstation. Two facts can be learnt from Table 2. Firstly, the centralised operations for creating filtering and refinement tasks take negligible time. Secondly, the spatial join cost is clearly dominated by the refinement cost, with the filtering cost takes less than 5% of the total CPU time. Thus, there is no doubt that the overall performance is critically dependent on load-balancing in the refinement phase for those spatial joins with relatively large number of candidates (recall that for the Sequoia join $|R| = 58411$, $|S| = 20974$ and the number of candidates is 84862). Although the load-balancing issue for the filtering phase becomes less important because of its very small total cost, the cell-based partitioning and bucket merger algorithms are still necessary as they not only provide the foundation for the whole framework by associating objects with cells, but also reduce significantly the key-pointer data duplicated among filtering tasks (as shown in Fig 10). In addition, they also reduce the number of candidates duplicated among filtering tasks to minimum (there are only 544 to 4193 inter-processor duplicate candidates for from 5 to 40 filtering tasks).

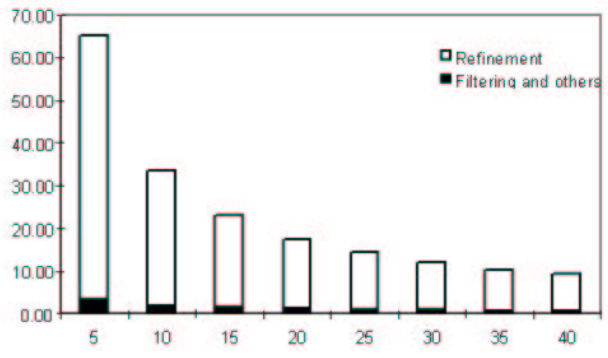

Figure 12: CPU cost (seconds) of parallel join processing with varying number of processors.

Fig 12 shows the CPU costs for the parallel spatial join algorithm against varying number of processors, where the costs for all pre-refinement operations are added up together. As Fig 12 illustrates, a near-linear CPU speedup has been achieved. This confirms the satisfactory performance of our cell-wise cost based refinement load re-balancing algorithm.

Object Name	Object Structure	Size (bytes)
bucket size/cell refinement cost	one number	4
key-pointer data	id, mbr, number of vertices	$6 \times 4 = 24$
full candidates	id1, id2, cell number, estimated cost	$4 \times 4 = 16$
data-request candidate	id, cell number	$2 \times 4 = 8$
full objects	id, n vertices as per object	$(n+1) \times 4$

Table 3: Sizes of the objects to be exchanged for parallel spatial join.

5.2 Communication Costs

Communication cost is measured by the total and the maximum per-processor amount of data exchange (object sizes are listed in Table 3). There are two types of data exchange: that between p_0 and the parallel processors for creating filtering and refinement tasks, and that among the parallel processors for redistributing key-pointer data, candidates and full objects. For the first type of data exchange, the amount of data is solely determined by the number of cells regardless the operand tables. For $N \times N$ cells, each processor sends no more than $N \times N$ numbers twice to p_0 for reporting non-empty small bucket sizes and for the estimated refinement cost of the full candidates in each cell; and p_0 broadcasts $N \times N$ numbers twice for the cell-to-task mapping for the filtering and refinement tasks.

Fig 13 shows the amount of the second type of data exchange for the Sequoia join with varying number of processors. Fig 13 (a) gives the total number of key-pointer data redistributed. Note that the total number of objects for the join is 79385 and for n processors the floor number of data exchange (when one object is assigned to one task) is $79385 \times (n-1)/n$ under the assumption of even and random initial data distribution (i.e., a minimum of 63508 and 77400 objects need to be exchanged for $n = 5$ and 40 respectively). Fig 13 (b) is the number of full and data-request candidates redistributed (local duplicate candidates are removed, but not the inter-processor candidates). The number of data-request candidates is close to the number of full candidates because of the random data distribution across processors. The number of inter-processor duplicate candidates increases, as expected, with the number of processors though it has been kept to very small by the bucket merger algorithm. Fig 13 (c) and (d) show the numbers and sizes (in Kbytes) of full objects transferred. As clearly illustrated in Fig 13, the amount of data exchange for all the three of the second type data redistribution operations increases with the number of processors but at a much slower rate, in contrast to fast decrease of CPU cost as shown in Fig 12. The benefit of minimising object duplication for the filtering phase can be seen by comparing Fig 13 with Fig 6, while the impact of our algorithm on the duplication of full objects has already been shown in Fig 10.

Fig 14 compares the amount for the first type and the three second type data redistribution operations. Although the amount of the first type data exchange increases linearly with the number of processors, its communication cost can be a constant because for many parallel computers the parallel processors can communicate with the interface processor p_0 directly and simultaneously. The total amount of data exchange for redistributing key-pointer data and candidates is very small in comparison to that for full object redistribution. It is clear that the major communication cost for parallel spatial join processing is for full object redistribution.

5.3 Response Time

Now let us consider communication bandwidth. Communication bandwidth of typical parallel database systems ranges from 10-100 MBytes per second [19]. The *effective* bandwidth can be lower because of the overhead associated with communication protocols and network traffic. Assuming the parallel processors are connected to a bus (thus the communication time is determined by the total amount of data exchange, not the maximum per processor amount of data exchange as for the parallel computers with parallel communication channels). All communication operations in our parallel join framework have been separated from local processing, thus the communication and local processing operations do not interfere with each other. Note that the CPU costs for pre-processing before data exchange, such as packing objects and removing duplicates, have been considered already in Section 5.1. Fig 15 shows the response time of parallel spatial join processing with varying number of processors and varying effective communication bandwidth from 0.1-10 MBytes/s. The

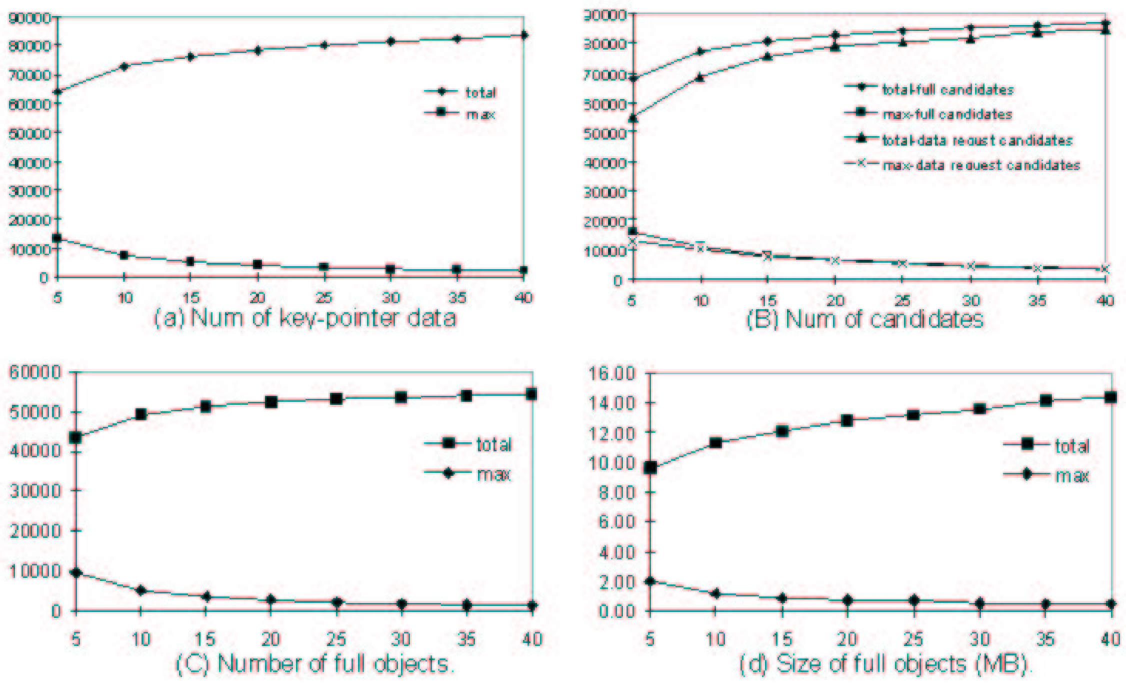

Figure 13: Full object redistribution costs with varying number of processors.

response time here is the sum of the maximum per-processor CPU time for the operations in Table 2 plus the estimated communication time for transferring the total amount of data shown in Fig 14. (Note that the possible overlapping of communication time among processors is not considered here.)

When effective communication bandwidth is higher than 5 MBytes/s, parallel spatial join processing becomes CPU-bound, thus it can be effectively parallelised to achieve the speedup close to that in Fig 12. When effective bandwidth is lower than 1 MB/s, our join algorithm is communication-bound with little response time reduction when the number of processors increases to more than 5; when effective bandwidth is as low as 0.1 MB/s, Fig 15 shows that the response time actually increases with the number of processors. This means that for a given spatial join operation a minimum communication bandwidth is required for efficient parallel processing. Note that the communication bandwidth requirements are mainly raised by redistribution of full objects. For most types of modern parallel computers, it is safe to conclude, at least for the join operations of sizes comparable to that considered in this paper, that the spatial join problem is CPU-bound, and that a near-linear speedup can be achieved. This conclusion holds for symmetric multiprocessor computers, most parallel database systems and even workstation clusters with Ethernet connections.

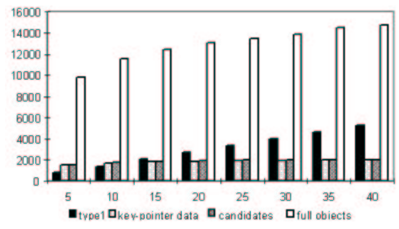

Figure 14: The amount (KBytes) of data exchange with varying number of processors.

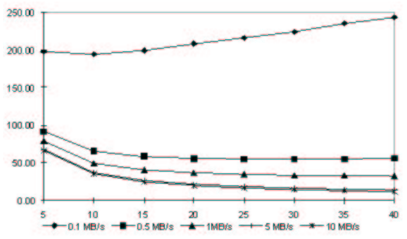

Figure 15: Response time (seconds) of the parallel spatial join algorithm with varying number of processors and different effective communication bandwidth.

5.4 Empirical Results

The Fujitsu AP-1000 is an experimental large-scale, single-user, MIMD distributed memory parallel computer developed by Fujitsu Laboratories, Japan [14]. Configurations range from 64 to 1024 individual processors connected by three separate high-bandwidth communication networks:

1. the B-net which is a 50 MB/s broadcast network which connects all parallel processors and the host, and is used for communication between the host and the processors. Using the B-net, the host can transmit data to all parallel processors simultaneously, and each processor can transmit data to the host.
2. the T-net which provides processor-to-processor communication. It is arranged as a two-dimensional torus in which each processor has links to its four neighbours in a rectangular grid. The bandwidth of each link is 25 MB/s. Wormhole routing is used so the processing of a processor is not interrupted when data are passed through it.
3. the S-net which supports synchronisation and status checking among parallel processors and the host.

The processors do not share memory. The AP-1000 is connected to and controlled by a host computer which is a Sun SPARCServer.

The computer we used in this paper is located in the Australian National University (see http://cap.anu.edu.au/cap). It consists of 128 processors, which are SPARC processor running at 25 MHz with 16 MB of RAM. Our parallel join algorithm is written in C, using library calls for communication over the networks. Each step in the join algorithm is synchronised.

Table 4 contains the response time, as well as the CPU costs for the hashing, filtering and refinement operations, for processing the Sequoia join on AP-1000 with varying number of processors. Due to limited RAM on each processor and our assumptions in Section 3.1, the minimum number of processors used here is 16. Note that a processor in the AP-1000 used here is not as powerful as the machine used for producing the simulation results in Table 2. The total time in Table 4 is the actual elapsed time, including the three CPU costs given in the table, all other CPU costs on both parallel processors and on the host, and all communication and synchronisation costs. Table 4 confirms the conclusions from the previous performance analysis using simulation data. It worth pointing out that the time for data exchange among parallel processors and the host is very small (a few seconds), and decreases as the number of processors increases.

#Procs	Hashing	Filtering	Refinement	Total
2×8	0.32	3.37	119.27	142.85
4×8	0.20	1.70	61.02	74.23
6×8	0.16	1.04	41.60	53.60
8×8	0.14	0.91	31.10	42.31
10×8	0.12	0.69	24.76	35.37
12×8	0.12	0.60	21.75	32.86
14×8	0.11	0.51	18.91	28.83
16×8	0.11	0.48	16.83	26.28

Table 4: The Sequoia join processing costs (seconds) on AP-1000.

6 Conclusions

In this paper we have proposed a framework for parallel spatial join processing, with algorithms given for each step of the framework to minimise both CPU- and communication-costs. Our framework based on the filter-and-refine strategy adopts the popular structure for data partitioning parallel relational join algorithms. We found that the MASJ data partitioning method is more suitable than the SAMJ method for parallel spatial join processing because it is simple, easy to be parallelised, and has a smaller number of MBR comparisons. Our method is very similar to conventional parallel join algorithms. It differs primarily in including new mechanisms to overcome the data skew problem caused by object duplication in both filter and refinement tasks.

Object duplication clearly imposes higher CPU and communication costs. Preservation of spatial locality is the key to minimising object duplication. This has been accomplished through a cell approach in both the filter and refinement steps.

The analysis of individual steps as well as the complete parallel spatial join process using the Sequoia data set has demonstrated a significant performance gain over other algorithms. We have shown that for the joins with a relatively high MBR join selectivity the response time is dominated by the refinement CPU cost. Using our algorithms, the spatial join operations can be effectively parallelised to achieve a near-optimal speedup for large joins in parallel environments.

Acknowledgment: The authors would like to thank Robert Power for many helpful discussions.

References

[1] D. J. Abel, V. Gaede, R. A. Power, and X. Zhou. Resequencing and clustering to improve the performance of spatial join. Technical report, CSIRO Mathematical and Information Sciences, Australia, 1997.

[2] D. J. Abel, B. C. Ooi, K.-L. Tan, R. Power, and J. X. Yu. Spatial join strategies in distributed spatial dbms. In *LNCS 951: Proceedings of 4th Int. Symp. on Large Spatial Databases (SSD'95)*, pages 346 – 367. Springer-Verlag, 1995.

[3] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing*, 24(1):1–13, 1983.

[4] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, 1993.

[5] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In *Proceedings of 12th International Conference on Data Engineering*, 1996.

[6] D. J. DeWitt and J. Gray. Parallel database systems: the future of database processing. *C. ACM*, 35(6):85–98, 1992.

[7] D. J. DeWitt et al. Practical skew handling in parallel join. In *Proc. 18th Int. Conf. on Very Large Data Bases*, pages 27–40, Vancouver, Canada, 1992.

[8] O. Günther. Efficient computation of spatial joins. In *Proceedings of 9th International Conference on Data Engineering*, pages 50–59, Vienna, Austria, 1993.

[9] R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–54, 1984.

[11] E. G. Hoel and H. Samet. Performance of data-parallel spatial operations. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 156–167, 1995.

[12] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[13] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 523–535, Barcelona, 1991.

[14] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. CAP-IID architecture. In *Proceedings of teh 1st Fujitsu-ANU CAP Workshop*, Kawasaki, Japan, 1990.

[15] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 210–221, 1990.

[16] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):66–74, 1983.

[17] M. L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 209–220, 1994.

[18] M. L. Lo and C. V. Ravishankar. Spatial hash-join. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 247–258, Montreal, Canada, 1996.

[19] H. J. Lu, B. C. Ooi, and K. L. Tan. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, 1994.

[20] J. Orenstein and F. A. Manola. Probe spatial data modeling and query processing in an image database application. *IEEE TOSE*, 14(5):611–629, 1988.

[21] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 259 – 270, Montreal, Canada, 1996.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: an introduction.* Springer-Verlag, 1985.

[23] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: a dynamic index for multi-dimensional objects. In *Proc. 13th Int. Conf. on Very Large Data Bases*, pages 3–11, 1987.

[24] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The SEQUOIA 2000 storage benchmark. In *Proceedings of ACM SIGMOD Int. Conf. on Management of Data*, pages 2–11, Washington, DC, 1993.

[25] X. Zhou. *Parallel Processing in Relational Database Systems.* PhD thesis, University of Queensland, 1994.