THERMAL-JOIN: A Scalable Spatial Join for Dynamic Workloads

Farhan Tauheed[†*], Thomas Heinis[¶*], Anastasia Ailamaki[‡]

[†] *Oracle Labs Zurich, Switzerland*

[¶] *Imperial College London, United Kingdom*

[‡] *École Polytechnique Fédérale de Lausanne, Switzerland*

ABSTRACT

Simulations have become ubiquitous in many domains of science. Today scientists study natural phenomena by first building massive three-dimensional spatial models and then by simulating the models at discrete intervals of time to mimic the behavior of natural phenomena. One frequently occurring challenge during simulations is the repeated computation of spatial self-joins of the model at each simulation time step. The join is performed to access a group of neighboring spatial objects (groups of particles, molecules or cosmological objects) so that scientists can calculate the cumulative effect (like gravitational force) on an object.

Computing a self-join even in memory, soon becomes a performance bottleneck in simulation applications. The problem becomes even worse as scientists continue to improve the precision of simulations by increasing the number as well as the size (3D extent) of the objects. This leads to an exponential increase in join selectivity that challenges the performance and scalability of state-of-the-art approaches.

We propose THERMAL-JOIN, a novel spatial self-join algorithm for dynamic memory-resident workloads. The algorithm groups objects in spatial proximity together into *hot spots*. Hot spots minimize the cost of computing join as objects assigned to a hot spot are guaranteed to overlap with each other. Using a nested spatial grid, THERMAL-JOIN partitions and indexes the dataset to locate hot spots. With experiments we show that our approach provides a speedup between 8 to 12× compared to the state of the art and also scales as scientists improve the precision of their simulations.

1. INTRODUCTION

Scientists no longer solely depend on studying a phenomena in their laboratory or in nature. They nowadays improve their understanding by first building three-dimensional spatial models and then by simulating the models at discrete intervals of time steps to gather key insight of the underlying principals that govern the phenomena. This enables scientists to make discoveries and even predict the behavior of phenomena, for example in cosmology [21] or seismology [1].

Computing the interaction between spatial objects that make up the model is crucial for many simulations. Doing so requires that the simulation identify at runtime all pairs of objects whose three-dimensional extents overlap. Identifying overlapping pairs of objects essentially translates into a spatial self-join that is performed repeatedly at each time step of the simulation. For example, to compute the gravitational force on a particular planet in n-body cosmological simulations [21], all other cosmological objects in proximity are retrieved using a spatial self-join.

During the simulation, the location of all spatial objects is changed at each time step to mimic the behavior of the phenomena studied, i.e., the locations of objects is read from main memory, manipulated and then updated in-place to reflect the latest state of the object. Two aspects of this problem introduce new challenges that limit the applicability of known approaches. First, all objects move together, i.e., the location of all objects is updated at every time step. All objects moving together is in stark contrast to use cases and applications, studied extensively in the literature, where only a subset of objects move at any instance in time. In case all objects move using an incremental join by re-using old results becomes unfeasible and executing a full join from scratch is more efficient. Second, unlike vehicular objects, the assumption that objects move in predictable trajectories for short distances does not hold and techniques involving trajectories or motion approximation thus cannot be used.

Recent analyses [34, 37] have demonstrated the limitations of existing join techniques for the problem of moving objects. Furthermore, known approaches do not scale in case scientists increase the precision of the simulation application. Increasing simulation precision is accomplished by increasing the number of objects in the dataset to reduce approximation errors due to spatial discretization and by increasing the threshold spatial region around the object to consider the effect of more objects in a wider spatial region. The consequence of both, i.e., increasing the number as well as spatial extent of the objects leads to an increasing join selectivity, which challenges known join techniques.

In this paper we present THERMAL-JOIN, an in-memory spatial self-join algorithm for moving objects. The approach is designed to address both the spatial aspect (high join selectivity) and the temporal aspect of the problem (massive and unpredictable updates). The key contributions are:

- THERMAL-JOIN leverages the dataset density to minimize the cost of joining. We introduce the concept of *hot*

[*] This work was done while the author was at EPFL.

SIGMOD'15, May 31–June 4, 2015, Melbourne, VIC, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.

http://dx.doi.org/10.1145/2723372.2749434.

spots, i.e., regions with high spatial density, where all objects are guaranteed to overlap with each other. Hot spots avoid the costly evaluation of overlap predicates. The benefits increase as the join selectivity increases, thus making the approach scalable.

- We use a novel linked-hash table to build, join and maintain a nested uniform grid. The rationale behind using a uniform grid is to index hot spots rather than to index spatial objects as is done in known grid-based approaches.

- The algorithm adapts and self-tunes the indexing structures used to account for the dynamic nature of the workload. We use a hill-climbing based approach that quickly converges without incurring an undue performance penalty.

Unlike previous studies we analyze the performance of competing approaches using three-dimensional datasets for both real (neuroscience) and synthetic benchmarks. We show that THERMAL-JOIN outperforms existing approaches while remaining competitive in terms of memory footprint. Most importantly, we demonstrate that THERMAL-JOIN scales with dataset size and object extent, two characteristics that are crucial to improve the simulation precision.

The remainder of the paper is structured as follows. In Section 2 we discuss related work and in Section 3 we motivate our work. In Section 4 we present our approach THERMAL-JOIN and discuss optimizations. We compare THERMAL-JOIN to related approaches in Section 5, analyze its performance in Section 6 and finally draw conclusions in Section 7.

2. RELATED WORK

Using our motivating application we review related work discussing how suited it is for workloads where the position of nearly all objects changes rapidly. We classify existing work based on whether it primarily serves for joining static spatial datasets (rebuilding and updating index structures at every time step) or if it has been designed for dynamic datasets, i.e., for joining moving objects. We do not distinguish whether approaches have initially been developed for main memory or disk as the ones developed for the latter can also be used in the former.

2.1 Iterative Static Spatial Join

One strategy is to use known static spatial join techniques [16] and update or rebuild their data structures from scratch at each time step before the join. Rebuilding the complete index or updating nearly all the objects at every simulation step is a considerable time investment that substantially slows down the simulation and is difficult to amortize over a join operation [36]. Both the nested loop join and the plane sweep join [29] thus do not maintain auxiliary data structures but at the same time their join process is prohibitively slow.

To improve the slow join process many join techniques use indexes. Updating the index, however, is very costly and it is typically cheaper to rebuild the index from scratch at every iteration (and to throw away the index after leading to short lived throw-away indexes [8]). Space-oriented partitioning indexes are particularly efficient in building an index as they use spatial grids or hierarchical space decomposition to index the objects. For example, PBSM [27] uses a uniform grid to partition the data and replicates objects based on how many partitions the object intersects with. Each partition is locally joined using a plane-sweep approach. Because objects are replicated, the same pair of objects may be tested multiple times, resulting in a substantial increase of intersection tests.

Hierarchical decomposition can be used to avoid replication as the widely used Octree [3] shows. The Octree and the MX-CIF Octree [15] are based on a uniform grid and split a cell uniformly if the number of objects in it exceeds a defined threshold. The objects in the split cell are assigned to the cells in which they do not intersect with a boundary (ideally to the children of the split cell). Similar to the Octree, S3 [19] uses a hierarchy of grids to index objects based on the smallest grid cell that fully encloses the object. The performance of both, S3 and Octree, suffers when objects are mapped to the root (or cells close to the root) of the hierarchical structure as they then have to be compared with all objects on lower levels, resulting in unnecessary intersection tests. The loose Octree [30] allows for a degree of imprecision so that objects can be assigned to lower levels when they intersect only slightly with a cell. The idea of using grids to parallelize the join has also been optimized for GPUs [33] as well as on a larger scale on the MapReduce framework[12, 23].THERMAL-JOIN as presented here is single threaded but can be parallelized like the aforementioned approaches.

THERMAL-JOIN also uses the idea of throw-away indexes based on space-oriented partitioning (grids) but attempts to reuse parts of the index whenever possible. The fundamental difference is that we do not use the grid to index the objects but rather to find the densely populated regions. Doing so enables us to avoid the problem of replication (every object is assigned to one grid cell only) and unnecessary intersections (objects are assigned based on the cells enclosing the center rather than the full object).

Data-oriented partitioning techniques avoid replication by dividing the space based on the distribution of objects (and assigning each object to one node only). The indexed nested-loop join [9] builds an R-Tree on one dataset and executes a range query on it for each object in the other dataset to find intersecting objects. The synchronous R-Tree traversal [5] joins two spatial datasets by traversing two R-Tree indexes built for each dataset from the roots to the leaf levels. While recursively traversing the trees, nodes on the same level are tested for intersection. The CR-Tree [18] optimizes the R-Tree for memory and targets at reducing cache misses and at reducing the index size (mainly through quantization). The optimizations generally improve performance by a constant factor over the regular R-Tree but do little to address the fundamental problem of overlap. In fact, quantization can also degrade performance as the approximated MBRs lead to more overlap. A further index based on the R-Tree and optimized for memory is TOUCH [26]. It reduces the number of overlap tests considerably and thus the join time but it is not designed for iterative changes to the dataset and the index has to be rebuilt in every iteration from scratch.

Data-oriented approaches like the R-Tree in general are prone to the problem of overlap and dead space [13], both of which severely degrade join performance. For our problem of datasets changing over discrete intervals of time, recent findings indicate that rebuilding the R-Tree and using the synchronous R-Tree traversal is the most efficient approach [34].

Several approaches based on a single grid level have also been developed for high-dimensional data (similarity join) and can be equally used for low dimensions. The Epsilon Grid Order [4] uses a high-dimensional grid of ϵ width and

chooses the order to join grid cells I/O-efficient, i.e., avoiding thrashing. The same ideas also helps in main memory to avoid cache pollution. Based on multiple grids and inspired by the quicksort algorithm, QuickJoin [17] uses a recursive partitioning technique and a nested loop join to join the results within each partition. DBSimJoin [32] builds on the QuickJoin technique and implements it as a non-blocking operator using an iterator interface.

For distributed map-reduce workloads, ClusterJoin [31] provides a framework for scalable similarity joins on skewed datasets. SSJOIN [2] addresses the set similarity join challenge by using locality sensitive hashing to compute the joins. Adopting similarity join techniques for spatial joins is feasible but they are not competitive compared to techniques designed for three-dimensional euclidean spaces as we show with experiments.

2.2 Joining Moving Objects

Instead of joining the dataset at each time step from scratch, spatio-temporal join methods [25] that are optimized for moving objects can be used. These methods join incrementally, i.e., reuse data structures built and used in previous time steps. Similar to static spatial indexes, joins can be performed using moving object indexes by querying the index for each object in the dataset. The ST2B-Tree [7] maps all objects on a uniform grid and indexes each object along with its identifier in a B+-Tree (cell identifiers are assigned based on a space-filling curve). OCTOPUS [36] on the other hand avoids an index and the associated maintenance cost when the data changes over time. It is, however, only applicable for mesh spatial datasets as it relies on the mesh connectivity to retrieve query results. The parallel implementation of a recently proposed moving object join [38] also uses a uniform spatial grid to index the locations of objects. The separation of the grid cells is exploited to use multiple threads to either update or join the data simultaneously.

To reduce the overhead of frequent maintenance proposed approaches [28, 35, 14] exploit the predictability in movement of the objects by approximating them with trajectories. The data structure used in the join therefore only has to be updated when the trajectory of the moving objects changes. Similarly, the adaptive two-level hashing approach [20] classifies objects according to their speed of movement. Slow moving objects are indexed with a fine-grained grid while fast objects are indexed with a coarse-grained grid. With our driving application, however, we cannot assume any predictability of the object movement and these techniques thus cannot be used.

3. MOTIVATION

Scientists no longer only rely on the study of a phenomena in their laboratory or in nature. Instead, they complement their understanding of the phenomena by building spatial models of it and by simulating the resulting models. In doing so they face considerable challenges in dealing with the data involved in simulations. One challenge they face is executing queries on fast changing spatial models used in simulations. Spatial simulations are used across many different scientific disciplines, from cosmology, medicine to material sciences. The development of THERMAL-JOIN is consequently driven by the needs of scientists who are facing performance bottleneck in simulating changes in massive spatial datasets.

In the following we first describe the scientific use cases and the problem in detail. We conclude this section by describing the novel data management challenges in the context of this problem.

3.1 Use Cases

The application motivating the development of THERMAL-JOIN originates from a collaboration with the neuroscientists in the Human Brain Project [24]. In the collaboration we study the performance bottlenecks and scalability of neural simulations. These simulations use part of the brain tissue represented by a collection of three-dimensional cylindrical spatial objects as shown in Figure 1(a). The simulations mimic the process of neural plasticity where the structure and connectivity of the cells change over time. In the first step of the simulation a distance join with the distance predicate d is executed to find pairs of objects within distance d of each other. The distance join is essentially performed by enlarging the spatial object by d in all dimensions and testing the enlarged extent for overlap. In the next step the electrical attraction and repulsion forces between pairs of objects are calculated to determine how the shape and connectivity of the neuron cells changes in subsequent simulation time steps.

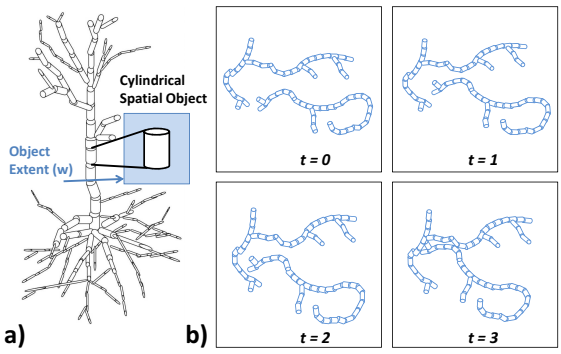

Figure 1: Cylindrical object representation of neurons with the spatial extent defined by the rectangle (a). Neural structure simulation at time t (b).

Finding pairs of overlapping objects efficiently is a challenge in simulation in general, as overlap or proximity between objects (representing parts of neurons, molecules or cosmological objects) usually defines how different spatial objects interact with each other. There are several interaction frameworks used by simulation applications that define how the models should be built and how the simulation can be discretized in space and time. The most widely used methods are smoothed particle hydrodynamics (SPH) [11] which are extensively used in computational fluid dynamics, N-Body simulations [3] used in particle and cosmology simulations and the Lennard-Jones pair potential method [10] used in molecular simulations. In all of these interaction frameworks the basic but also most crucial task is to access all pairs of overlapping objects which in the case of neural simulation takes 66% of the total time (using an octree based join technique). This task is performed several times during the simulation as the objects change their location over time and the problem thus translates into an iterative spatial self-join in main memory.

3.2 Iterative Spatial Self-Join

We first discuss the static (spatial) aspect of the iterative spatial self-join challenge and then describe the dynamic (temporal) aspect. Consider a spatial dataset D with N

three-dimensional spatial objects. As is standard practice in related work [16], we use the minimum bounding rectangle (MBR) as the spatial extent w_i for each spatial object $s_i \in D$. The problem of a spatial self-join is to find all pairs of spatial objects $(s_i, s_j) \in D \times D$, such that the spatial object pair satisfies the predicate of spatial overlap, i.e., $overlap(w_i, w_j) > 0$. The self-join result does not allow reflexive object pairs (s_i, s_i) and counts commutative object pairs $[(s_i, s_j), (s_j, s_i)]$ as one join result. In scientific simulations the spatial extent w_i (also known as cut-off radius or ϵ) is the same for each object in the dataset and represents a region where an object might interact with another object, e.g., a gravitation field.

During the simulation the location of each spatial object is changed (updated in-place) by the simulation application to mimic the behavior of the phenomena studied, making the problem of a spatial (self-) join more challenging. The spatial extent w_i and number of objects remain constant for a during simulation (but can change for different simulation runs) because each object represents a physical entity (planets or stars) that remains the same during the course of the simulation (as shown in Figure 1(b)). The changes are applied iteratively at discrete intervals that affect every object in the dataset. This is different from previous work [28, 14] where only a subset of objects change location or velocity.

Our goal is to design a general solution, not limited to a particular simulation application. We therefore treat the simulation application as a black box and do not rely on the update mechanism of the spatial objects to optimize the approach. The changes to location of the objects are therefore treated as unpredictable. This means known approaches based on time parametrization and predictable trajectories [28, 35] are ill-suited.

3.3 Data Management Challenge

Even for reasonably small dataset sizes (a few gigabytes in main memory) an iterative spatial self-join can take hours to complete and it therefore creates a substantial bottleneck in simulation applications. To make matters worse scientists want to increase the precision of their simulations by first, increasing the spatial dataset size (to increase the spatial resolution of the models) and then increasing the three-dimensional objects' extent (to consider more surrounding objects to compute the interaction). Both, increase in dataset size and object extent, make simulations more realistic but at the same time they also increase join selectivity resulting in poor scalability of known approaches for self joins.

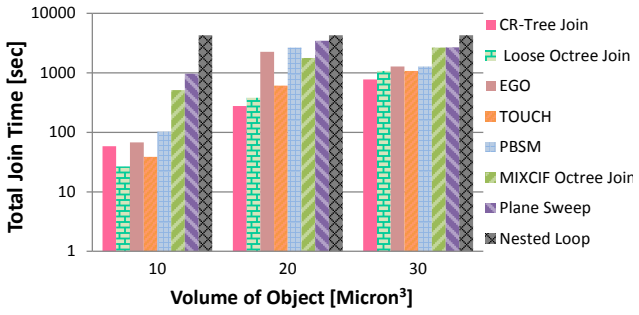

Figure 2: Effect of join selectivity on self-join time.

To demonstrate the effect of join selectivity we perform an experiment using a small neural dataset consisting of one million objects. Figure 2 shows the results of a self-join for one time step of the simulation by comparing existing in-memory join techniques. We change only the volume of the objects extent w from $10\mu m^3$ to $30\mu m^3$ to increase the join selectivity. As the experiment shows, the join time increases by one order of magnitude and the performance of known methods approaches the prohibitively slow nested-loop join.

Existing approaches reduce the join time by minimizing the unnecessary overlap tests between objects that are far apart in space. If, however, the join selectivity increases, the "necessary" overlap tests between objects that satisfy the join predicate increase quadratically, making known approaches inefficient for datasets with high join selectivity.

4. THE THERMAL-JOIN APPROACH

THERMAL-JOIN, the approach we develop, addresses the problem of high join selectivity by organizing the dataset into *hot spots*, i.e., regions of very high spatial density. The intuition behind THERMAL-JOIN is to process a self-join within each hot spot as efficiently as possible while minimizing the overhead of joining objects of a hot spot with objects in its surrounding spatial region. A hot spot is a region in which all objects are guaranteed to overlap with each other and it thus makes pair-wise overlap tests among objects that are in the hot spot unnecessary. The proposed join strategy used in THERMAL-JOIN is therefore fundamentally different to existing join approaches where the focus has been to provide better join performance by minimizing the unnecessary overlap tests between objects far apart in space.

Finding hot spots in spatial datasets in simulations can become expensive as the dataset changes unpredictably at every time step of the simulation. We use a two-level nested spatial grid to do so efficiently. In contrast to existing grid-based solutions, the primary purpose of using the grid in THERMAL-JOIN is to provide efficient access to hot spots instead of using the index structure to query the dataset and to locate objects in spatial proximity of each other. The choice of using a spatial grid further favors efficient rebuilding and maintenance as the dataset changes during the simulation.

In the following we describe the THERMAL-JOIN approach in detail and highlight the key insights it uses for self-joining dynamic spatial datasets. We divide the approach into three phases:

1. **Index Building:** In Section 4.1 we describe how to build the spatial grid structure using a linked-hash table data structure that provides efficient access to the hot spots and also to the neighboring regions while keeping the memory footprint low.

2. **Joining:** In Section 4.2 we describe the join algorithm that uses the grid index. We divide the joining phase into two parts, first joining objects within each grid cell (internal Join) and then each grid cell with adjacent cells (external Join).

3. **Index Maintenance:** As the dataset changes during simulation we tune the grid index to provide better join performance. In Section 4.3 we describe how to find an efficient configuration for the grid index and also a mechanism to recycle grid cells to further minimize the index building/maintenance time and memory footprint.

4.1 Index Building

The index building of THERMAL-JOIN requires efficient means to build and maintain spatial grids. In the following we describe in detail the spatial grid indexes used along with

the linked-hash data structure that we use to implement the grid index and to avoid both the problem of overlap and that of empty grid cells.

Join algorithms based on data-oriented partitioning like variants of the R-Tree [5, 26] suffer from the overhead of overlapping nodes in the tree structure and high index building cost. To avoid the issue of overlap we partition the datasets using space-oriented partitioning or, more precisely, by using a uniform spatial grid. The spatial objects of the model dataset are mapped to the grid based on their center and therefore are not replicated unlike space-oriented join techniques like PBSM [27]. Avoiding replication is pivotal as it makes the technique scale with the increasing spatial density of the dataset joined.

Real simulation datasets have a skewed data distribution that cause the majority of the grid cells to remain empty. Managing empty cells is crucial because it can be a substantial overhead in terms of storage space required. For example, just using a null pointer to indicate an empty cell for a rather coarse resolution 3D grid with $(1000 \times 1000 \times 1000)$ cells requires 7.5GB of space in memory. Our approach uses a hash table that only keeps cells that have at least one object assigned to it. This reduces the memory consumption significantly but the cost of accessing (spatially) neighboring cells during the join phase increases as hash lookups are required which cause a significant overhead due to collisions (particularly in the case of fine grained grids).

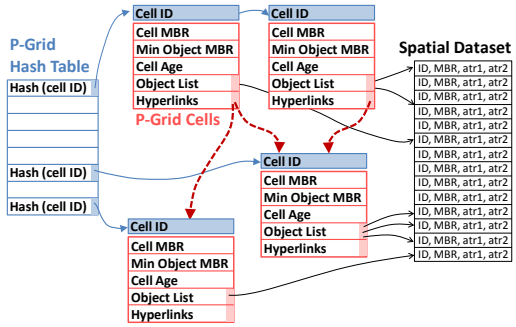

Figure 3: Linked-hash table data structure.

THERMAL-JOIN addresses the problem by using a linked-hash table that provides direct access to neighboring cells through pointers (hyperlinks) introduced in the index building phase as shown in Figure 3. This means we can use the same hash table for both index building and join phase. During index building hash lookups are used to map objects to their respective cells while in the more costly join phase the hyperlinks are used to avoid hash lookups when accessing neighboring cells.

THERMAL-JOIN uses a two-level nested grid. The primary (permanent) grid, or simply P-Grid, is built and maintained to reflect the most recent location of each object for the last time step. For each cell of the P-Grid it is possible to further divide the space using a temporary throw away grid T-Grid to enhance join performance. Crucially, as opposed to many hierarchical grid [19] or tree based techniques [30], the resolution of each T-Grid built for each cell of P-Grid can be different and in many cases a T-Grid is not even required to further organize a cell of the P-Grid.

The P-Grid is built by calculating the cell each object belongs to. The assignment is based on the information of what cell the center of an object falls into. Once the unique identifier of the cell containing the object is obtained, the identifier is used to perform a hash lookup to determine if a new cell is needed or if the object pointer can be assigned to an existing cell's object list. Once all the objects are assigned to their grid cells, the object lists are sorted along the x-dimension to facilitate the join phase. Furthermore, for each cell we find all the neighboring adjacent cells and add their pointers to the hyperlink list to provide efficient access to neighboring cells. The algorithm used to build a P-Grid is described in pseudocode in Algorithm 1. The temporary T-Grid is built during the join phase for each cell. We will discuss this process in Section 4.2.

The algorithm described terminates by returning a P-Grid hash table with all objects assigned based on their centers and each cell linked to neighboring cells. This algorithm is used for building the P-Grid from scratch. Its performance can be further improved by re-using cell structures during index maintenance, described in Section 4.3.

Algorithm 1 THERMAL-JOIN: Index Building Phase

Input: Dataset: Spatial model dataset
Data: PGrid: Empty hash table of cells
Output: PGrid: Hash table with objects assigned
foreach $object \in Dataset$ **do**
  $center \leftarrow object.MBR$
  $cellID \leftarrow calculateCellID(center)$
  $targetCell \leftarrow PGrid.hashlookup(cellID)$
  if $targetCell.found()$ **then**
    $targetCell.objectlist.insert(\&object)$
  else
    $cell \leftarrow newCell()$
    $cell.objectlist.insert(\&object)$
    $PGrid.insert(cellID, \&cell)$
  end
end
foreach $cell \in PGrid$ **do**
  $cell.objectlist.sortX()$
  $neighborsList \leftarrow getNeighbors(cell)$
  foreach $n \in neighborList$ **do**
    $neighborCell \leftarrow PGrid.hashlookup(n)$
    $cell.Hyperlinks.insert(\&neighborCell)$
  end
end
return $PGrid$

4.2 Joining

Once the P-Grid is constructed the join phase can start. It works by processing each P-Grid cell from the hash table in no particular order. The algorithm works in two parts; first, joining objects in each cell with its adjacent P-Grid cells (external join) and then joining all objects within each P-Grid cell (internal join).

4.2.1 External Join

Assigning objects based on the center has the advantage that every object is assigned to only one P-Grid cell, thereby avoiding object replication. In order to retrieve accurate join results, however, the objects assigned to each cell need to be joined with the objects assigned to adjacent grid cells as well. To avoid any pair of cells being considered for external join more than once we only consider half of the adjacent cells. The number of adjacent cells taken into account for the external join depends on the P-Grid cell width and the width of largest object in the dataset. For example, if the

P-Grid resolution is set such that the width of the cell is equal to the width of the largest object then we only need to consider half of the immediate adjacent cells, e.g., only the cells located in northwest, north, northeast, east direction as shown in Figure 4(a) (as a two-dimensional illustration). For three-dimensional datasets the number of adjacent cells is 13. In case the P-Grid resolution is set such that the width of the cell is less than the width of the largest object in the dataset then multiple layers of adjacent cells need to be considered for the external join as shown in Figure 4(b). The number of layers of adjacent cells can be calculated by dividing the P-Grid cell by the width of the largest object.

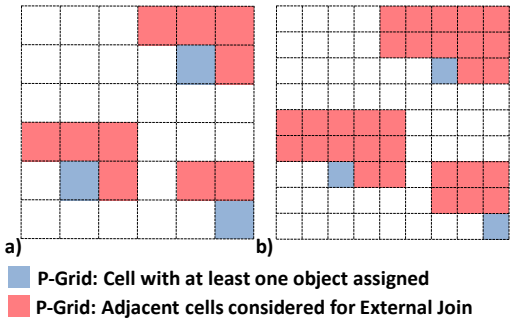

Figure 4: Adjacent cells used in the external join, (a) when P-Grid cell width = largest object width, (b) when P-Grid cell width ≤ largest object width.

The $getNeighbors(x)$ function in Algorithm 1 computes the cell identifiers of half of the adjacent cells in order to introduce hyperlinks that provide efficient access to these adjacent cells. THERMAL-JOIN determines the size of the largest object in the dataset while loading the dataset.

Once the index is built and the hyperlinks are created, we join each object a_i assigned to every cell A in the P-Grid with all the objects b_{jk} in each adjacent cells B_k of A. The join is performed using the plane-sweep approach which explains why we sorted the object list of each cell on the x-dimension during building as described in Algorithm 1. We use an optimized variant of plane-sweep approach, i.e., if the MBR of any object a_i encloses the entire MBR of cell B_k then we can avoid expensive overlap tests for this object during the external join and directly report that all $b_{jk} \in B_k$ overlap with a_i. This is because the MBR of cell B_k encloses the center of all objects assigned to it and therefore every object in B_k is guaranteed to overlap with a_i.

4.2.2 Internal Join

The concept of *hot spots* is central to THERMAL-JOIN. We define a hot spot as a grid cell, whose width in dimension k is equal to or less than the width of the smallest object assigned to that cell in dimension k (smallest object with respect to dimension k). By choosing the width of the cell less than or equal to the width of the smallest object we can ensure that irrespective of where the centers of the objects are located inside the cell, all objects will overlap with each other and therefore expensive pair-wise overlap tests can be avoided, thereby substantially accelerating the internal join.

Naturally, designing the P-Grid such that each cell is equal to the width of the smallest object in the entire dataset means that all cells are hot spots. However, if the dataset contains only a few very small objects this strategy forces the grid to have a very fine resolution. Doing so speeds up the internal join but will also increase the overhead for the external join because smaller cells mean that more adjacent

Algorithm 2 THERMAL-JOIN: Joining Phase

Data: PGrid: Hash table with objects assigned,
TGrid: Empty array of cells
Output: results: List of object pairs (result of the join)
foreach $cellA \in PGrid$ **do**
  foreach $cellB \in cellA.Hyperlinks$ **do**
    $results \leftarrow PlaneSweep(cellA, cellB)$
  end
  if $cellA.MBR \leq cellA.minObjectMBR$ **then**
    $results \leftarrow allCombinations(cellA.objectlist)$
  else
    $TGrid.initialize(cellA.minObjectMBR)$
    foreach $o \in cellA.objectlist$ **do**
      $center \leftarrow o.MBR$
      $id \leftarrow calculateCellID(center)$
      $TGrid.cellArray[id].objectlist.insert(\&o)$
    end
    foreach $subCell \in TGrid$ **do**
      $results \leftarrow allCombinations(subCell.objectlist)$
      $neighborSubCells \leftarrow getNeighbors(subCell)$
      foreach $neighbor \in neighborSubCells$ **do**
        $results \leftarrow PlaneSweep(subCell, neighbor)$
      end
    end
    $TGrid.clear()$
  end
end
return *results*

cells need to be considered as shown in Figure 4. Making the P-Grid cell larger than the smallest object, on the other hand, requires fewer adjacent cells to be considered but not all P-Grid cells satisfy the condition of a hot spot and the internal join may thus take longer. Setting the P-Grid resolution introduces an interesting trade-off that we will discuss in Section 4.3.

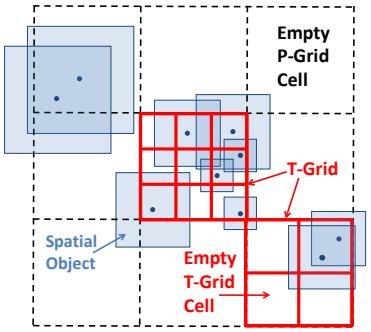

Figure 5: Nested T-Grid built for each P-Grid cell.

If any P-Grid cell is a hot spot then the join results can be directly reported by generating all possible pair-wise combinations (without overlap test) for objects assigned to that grid cell. The case where a P-Grid cell is a hot spot is shown in Figure 5 (top left cell).

Joining the objects in a P-Grid cells that are not hot spots is more challenging. The objects assigned to the same P-Grid cell are densely packed together with a considerable chance (but no certainty) that all of the objects will overlap with each other. Using a plane-sweep approach to join inside the P-Grid cell, as we use for the external join, is not efficient anymore, because the plane-sweep join degenerates into a nested-loop join given the high object density.

We repeat the process of creating hot spots within P-Grid cells that do not satisfy the condition of hot spots themselves. Objects assigned to the same P-Grid cell are likely to have a smaller variation in terms of object width compared to the entire dataset and we can thus design the nested grid T-Grid such that the width of each cell in T-Grid is exactly equal to the width of the smallest object assigned to the P-Grid cell. Each P-Grid cell that is not a hot spot can therefore have a different resolution for the sub grid (T-Grid) as is shown in Figure 5.

The resulting cells in the T-Grid are certain to be hot spots and consequently no further recursive subdivision is necessary. Similar to joining the P-Grid we join objects assigned to the T-Grid in two phases, i.e., first joining objects between two different T-Grid cells using an optimized variant of the plane-sweep approach described in Section 4.2.1, followed by a quick internal T-Grid cell join by simply reporting all pair-wise combinations of objects assigned to the cell as join results. The implementation we use for the T-Grid is different from the P-Grid. We use an array to manage the grid cells instead of a linked-hash table because the T-Grid in practice has only a few cells and therefore the space overhead of representing empty cells is insignificant. Moreover, using an array implementation makes building the T-Grid very fast. This means we can now build the grid on demand and throw it away after we have processed the internal join for a P-Grid cell as described in Algorithm 2.

4.3 Index Maintenance

The discussion of THERMAL-JOIN so far has focused on a static scenario. To make use of THERMAL-JOIN for dynamic, i.e., changing datasets, the grid needs to be updated at runtime, e.g., of a simulation application. We first discuss an incremental index maintenance strategy and then describe a mechanism to adaptively tune the index resolution at each time step of the simulation.

4.3.1 Incremental Index Maintenance

Performing the spatial self-join iteratively requires that the index reflects the recent location of the objects at every time step. A straightforward approach like building the index from scratch for each time step using the method described in Section 4.1 introduces considerable overhead. Using an incremental approach instead is likely to yield substantially better results. Existing strategies like join techniques based on predictable trajectories or techniques that assume that only a small subset of objects change their location, will not work as we cannot assume to be able to predict trajectory or location of all objects during simulation.

In THERMAL-JOIN we implement the idea of incremental maintenance by re-using parts of the P-Grid index. At each time step every object's location is inspected and the object is assigned to a new P-Grid cell (if needed). Time in this process can be saved by recycling data structures: for objects assigned to the same P-Grid cell or to a non-empty neighboring cell no new cells and hyperlinks need to be created. The performance benefits of recycling data structures increase considerably for fine resolution grids where many cells need to be created.

In order to implement the incremental maintenance approach Algorithm 1 only needs to be changed so that for each time step the P-Grid is not deleted but only the *objectlist* of a cell is cleared before assigning objects based on the last time step.

This means that if all the objects in a cell migrate away from it, the vacant cell will still exist in the P-Grid hash table. This becomes useful in case any object in the future moves and tries to reuse the vacant cell. In the long run, however, the memory footprint of the P-Grid will increase and keeping empty cells will become unsustainable. To address this issue we use a garbage collection approach to prune out the vacant cells. A garbage collection does not need to run at every time step instead it can be triggered using a simple policy, e.g., garbage collection is performed if the number of vacant cell exceeds a defined threshold, e.g., the number of vacant cells exceeds 35% of the total number of cells of the P-Grid.

4.3.2 Index Tuning

Grid-based join techniques are challenging to configure (i.e., to set the grid resolution) for optimal performance. One approach is to develop and use an analytical model based on the workload and the compute resources to configure the grid. Doing so requires sampling the workload to capture the data distribution and also requires in-depth knowledge about the hardware. THERMAL-JOIN, on the other hand, uses a more pragmatic method and iteratively tunes the grid resolution during the simulation.

The idea of iterative tuning makes the assumption that during the simulation the spatial distribution (not the individual location of objects) of the workload does not change drastically between subsequent time steps. In other words, once tuned, the THERMAL-JOIN configuration can be used for several simulation time steps efficiently and the overhead of tuning can thus be amortized over several time steps. This assumption works for real world workloads, for example, in cosmological simulation even though the objects (stars) change their location the distribution (clustering in galaxies) does not drastically change between two time steps.

In THERMAL-JOIN we use two grids. The resolution of the T-Grid is always fixed such that the width of the cells is equal to the width of the smallest object in the dataset. The resolution of the P-Grid can vary and the performance of THERMAL-JOIN depends on configuring this grid properly. We discuss the P-Grid resolution using a normalized metric r, where $r = 1$ means that the grid resolution is fixed (cell width is equal to the width of largest object in the dataset).

To understand the trade-off associated with the P-Grid resolution, let us assume that all objects are of equal size. If we set the resolution of the P-Grid such that $r > 1$ then the cost of performing the internal join increases because the P-Grid cell no longer is a hot spot. Using a fine resolution grid with $r \leq 1$, on the other hand, means that the cost of internal join is reduced but at the same time the number of cells required in the external join of the P-Grid increases drastically, thereby increasing the cost of external join and of building the index. The discussion implies that the function modeling the performance of THERMAL-JOIN at time step t is a convex function $F_t(r)$ of resolution r with a single global minima at the optimal resolution r_{opt}.

The sweet spot r_{opt}, however, is different for different workloads. Figure 6 shows the trade-off by experimenting with THERMAL-JOIN on four synthetic (uniform random distribution) datasets containing 10 million objects each. The join selectivity is increased by increasing the object width in each dataset. The experiment shows the result of performing self-join for one time step (static) to highlight the convex performance function with different r_{opt}.

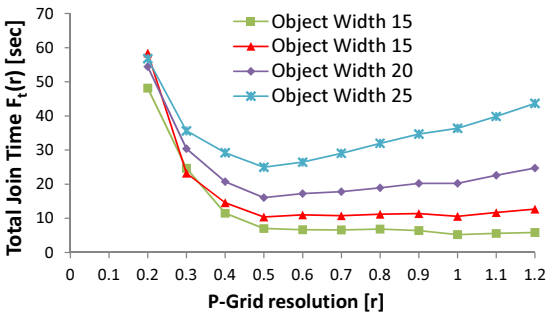

Figure 6: Impact of resolution r on join $F_t(r)$.

Our goal for tuning THERMAL-JOIN is not to find the optimal r_{opt} but to find r' with reasonable overhead such that $F_t(r') \approx F_t(r_{opt})$. We thus use a hill climbing approach to iteratively search for r' starting with $r_1 = 1$ and define the convergence based on Equation 1.

$$|F_{n-1}(r_{n-1}) - F_n(r_n)| \leq threshold \quad (1)$$

Each time we use a different r_n we need to rebuild the P-Grid from scratch leading to a substantial overhead in terms of building time. During the iterative hill climbing we may encounter r_n such that $F_n(r_n)$ is very high causing additional overhead. In practice, however, we experience that the performance spikes only appear in the first few iterations of hill climbing when the algorithm is trying to explore the performance function F.

Already with a threshold of 10% we see that the tuning process converges quickly, i.e., typically in 6-8 time steps. Once the convergence is achieved we use r' for subsequent time steps by turning off the iterative tuning. In case the distribution of the workload changes during the simulation, we need to re-tune P-Grid by turning the tuning on again. THERMAL-JOIN does so in case of significant performance change during simulation as described by Equation 2.

$$|F_{n-1}(r') - F_n(r')| > threshold \quad (2)$$

5. EXPERIMENTAL EVALUATION

In this section we first describe the experimental setup and demonstrate the benefits of THERMAL-JOIN by using real neural simulation workload. Furthermore, we perform a sensitivity analysis using a synthetic moving object benchmark to understand the performance by changing one workload variable at a time. Before we conclude we also analyze the THERMAL-JOIN algorithm in depth by discussing a breakdown of the performance and the impact of index tuning.

5.1 Setup and Methodology

The experiments are run on a Linux Ubuntu 2.6 machine equipped with 2x Intel Xeon Processors each with 6 cores running at 2.8GHz, with 64kb L1, 256KB L2 and 12MB L3 cache and 48GB RAM at 1333MHz. The storage consists of 2 SAS disks of 300GB capacity each. In the following we discuss the software setup as well as the configuration details for the competing approaches.

5.1.1 Software Setup

Each join algorithm implemented uses a single CPU core to ensure a fair comparison. We do not use the available implementations provided in [34] because the code assumes two-dimensional spatial datasets and extending it to support the three-dimensional dataset we use is not trivial [37]. The implementations are all written in C++.

The simulation datasets are loaded in memory and organized as a list of spatial objects that includes the minimum bounding rectangle (MBR) of each object, identifiers as well as attributes related to the properties of the object (e.g., electrical conductivity, mass, etc.). The simulation software processes the list at each iteration, i.e., it reads the objects, changes their location and then it writes the new location of each object in-place.

None of the join algorithms rearranges the list of spatial objects because the simulation application assumes the object to be in a particular order. Every approach therefore uses pointers to objects in the list to inspect the current location of the object (similar to Figure 3). The self-join is performed (atomically) at every time step only after the simulation application has completed modifying the list of spatial objects. The dataset in main memory is therefore always in a consistent state when the self-join is executed.

5.1.2 Competing Approaches

We use the fastest in-memory self-join approaches to compare against THERMAL-JOIN based on the results we obtained in Figure 3. First we use the synchronous R-Tree traversal which has recently been identified as the fastest in-memory join approach [34]. Second, we use a grid based join technique Epsilon Grid Order (EGO) [4] proposed for n-dimensional similarity joins and finally, we also include two recent approaches, the in-memory join algorithm TOUCH [26] and the join algorithm based on the space-oriented loose Octree [30]. In the following we discuss the configuration of each approach:

- CR-Tree: We implement the synchronous R-Tree join using a cache conscious variant of the R-Tree. Using the CR-Tree [18] not only improves performance but also reduces the memory footprint by using quantized tree nodes. To perform a self-join the tree is built only once using the STR R-Tree [22] bulkloading technique, while two pointer based synchronized breadth-first-search traversals are used to implement the algorithm. We perform a parameter sweep to determine the best tree fan-out (i.e., 11) for our workloads.

- TOUCH: Similar to the CR-Tree join, we use a parameter sweep to find the best fan-out (i.e., 2) for our workloads.

- Loose Octree: We build an indexed nested-loop variant of the join algorithm over a Loose Octree index that promises fast index building. After building the index the same dataset of spatial objects is used as range queries to find all overlapping object pairs by taking care not to report reflexive and commutative pairs. By performing a parameter sweep we found that the looseness factor of $p = 0.1$ yields the best join performance.

- EGO: We implement the epsilon grid order join for three-dimensional in-memory workload. The grid resolution (epsilon) is based on the object size used in the dataset.

- THERMAL-JOIN: we do not need a parameter sweep to configure the resolution of the P-Grid as the algorithm self-tunes at runtime. We use 10% as the convergence threshold to terminate the hill-climbing approach. Garbage collection is triggered if the vacant cells exceed more than 35% of the total cells.

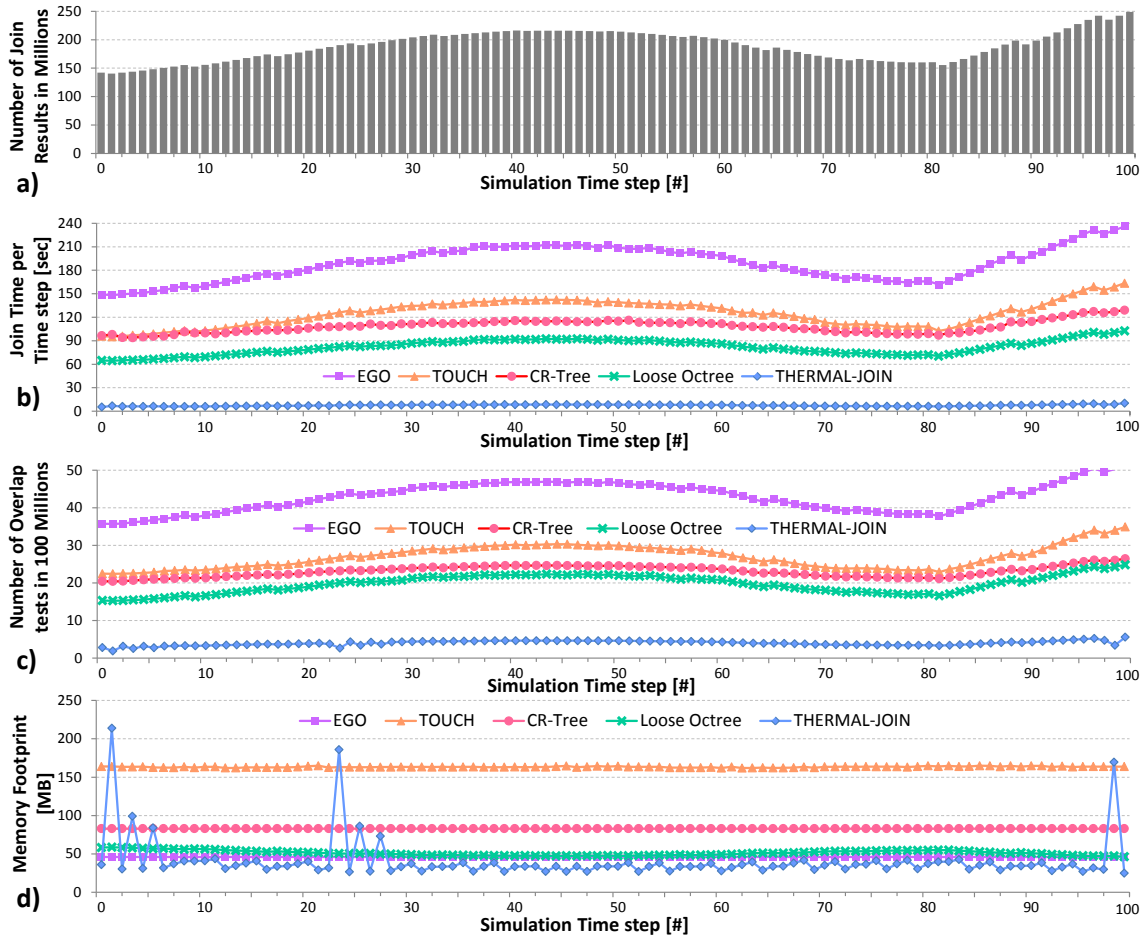

Figure 7: Neural simulation for 100 time steps. (a) Join selectivity in terms of number of join results, (b) total join time per time step, (c) total overlap tests per time step and (d) memory footprint per time step.

5.2 Neural Simulation

We use a real neural simulation workload as described in Section 3.1 for comparing the performance of THERMAL-JOIN with existing approaches. The dataset represents a small part of the rat brain tissue containing 1692 neurons. The full dataset contains the branches of the neurons (as illustrated in Figure 1) modelled with 4 million cylindrical objects amounting to 160 MB on disk. Already with a small dataset like this the self-join can take several hours as we show with experiments. We fix the object extent to $15 micron^3$ and present the results for the full 100 time step simulation. Subsequently we study the scalability of the competing approaches by increasing the dataset size as well as object extent.

5.2.1 *Full Simulation*

During the simulation the objects change their location which can also lead to a change of the distribution of the spatial objects. This affects the join selectivity: if, for example, the spatial density of a region increases, the number of join results for a time step increases as well as is shown in Figure 7(a). In Figure 7(b) we can see how existing approaches are very sensitive to the join selectivity because the number of overlap tests required dominates the join time (takes more than 95% of the join time and increases with the selectivity).

The join based on the loose Octree provides the second best performance for this dataset. THERMAL-JOIN on the other hand provides a further speedup from 9.4× to 11.1× when compared to the loose Octree. Crucially, THERMAL-JOIN is the least sensitive to the join selectivity thanks to the concept of hot spots that reduce the number of overlap tests as corroborated by Figure 7(c).

In terms of memory footprint, EGO, CR-Tree and TOUCH remain unaffected by the join selectivity as shown in Figure 7(d). Both CR-Tree and TOUCH approaches use variants of the R-Tree algorithm where the number of nodes in the tree depends on the dataset size rather than the distribution. The CR-Tree, however, uses quantized MBRs that reduce the space required for each node and it is thus more space efficient than TOUCH. EGO simply builds a spatial grid where objects are assigned to at most one grid cell similar to Octree techniques but no hierarchical structure is used, making it memory efficient. The nodes of the loose Octree [30] do not have a fixed number of objects assigned to them and as the distribution of the dataset becomes denser in certain areas, more objects can be assigned to the same node, thereby decreasing the number of nodes and the memory footprint when the join selectivity increases.

For THERMAL-JOIN we observe considerable spikes of memory consumption which is due to the index being tuned during at certain time steps. This tuning is not done once because the dataset changes distribution considerably during the simulation, triggering the tuning process three times in 100 time steps. The hill-climbing method used for tuning

the P-Grid often explores the option of using very fine grid resolutions that increase the number of cells required and therefore increase the memory footprint. Apart from the spikes, it appears that the memory footprint measurement resembles a saw-tooth function. This is because of incremental building of the P-Grid and then periodic garbage collection to deallocate vacant cells.

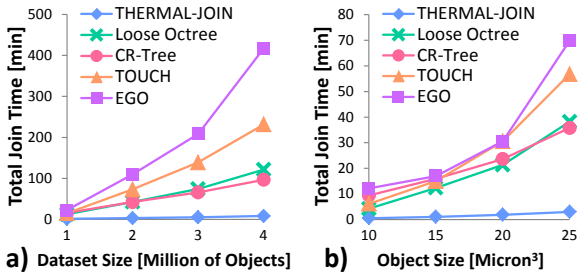

Figure 8: Scaling-up neural simulations in terms of (a) dataset size, (b) object width.

5.2.2 *Scalability*

We study the scalability with shorter duration simulations, i.e., only until the first 10 time steps to measure the join time for larger and more selective neural datasets. In Figure 8(a) we increase the dataset size based on the neural samples provided by the neuroscientists where the larger neural datasets are built by increasing the number of neurons (roughly 420 neurons resulting in approximately 1 million objects) considered for each experiment. In this experiment the object extent (or object width) is fixed to $15 micron^3$. Increasing the number of spatial objects used in the neural simulation in the same space increases the density of the dataset and thus also increases the join selectivity.

In this experiment THERMAL-JOIN outperforms competing approaches by a factor of 11.4× to 12×. Interestingly, although the loose Octree performs better for the dataset with one million objects, it scales poorly compared to the CR-Tree. This is because the join using the loose Octree uses a simple indexed nested-loop while the CR-Tree benefits from the synchronized traversal method as the dataset size grows. Although EGO uses a very efficient grid-based index, the join uses a nested loop join and therefore does not scale as the number of objects increase in each grid cell.

In Figure 8(b) we increase the object extent and keep the number of objects fixed to one million objects. In this experiment the CR-Tree again scales better than the loose Octree as the join selectivity increases. For THERMAL-JOIN the speedup increases from 8× to 12× suggesting that its benefit will further increase with increasing join selectivity.

5.3 Synthetic Benchmark

Experimenting with real workloads is essential to assess how approaches behave in real world scenarios. When doing so, however, understanding the impact of each individual workload characteristic on the performance is difficult. We therefore use a synthetic benchmark to isolate each characteristic and perform a sensitivity analysis by changing six different workload characteristics.

We use the benchmark described for moving object datasets [6] to calculate iterative self-joins. Unless stated, we use the following default workload characteristics of the benchmark. We use a uniform random distributed benchmark with 10 million three-dimensional spatial objects (with a size of 400MB) inside a boundary defined by (0,0,0) and (1000,1000,1000). Each object has a fixed object width of 15 units and the simulation lasts for 10 time steps. During the simulation each object is assigned a uniform random motion vector that is used to translate the object by 10 units at each time step. If the objects intersect with the boundary of the spatial extent the motion vector is inverted to ensure that the spatial boundaries remain the same.

Similarly, a second benchmark is created representing a skewed workload with 10 million objects and 15 units of object width. The skewed distribution is created using a normal distribution with the center of the cluster chosen randomly (uniform) and a spread defined by the standard deviation $sd = 1$. All objects within the skewed cluster have the same motion vector so that during simulation the distribution is preserved.

In the following experiments we use the uniform benchmark to understand the impact and benefits of THERMAL-JOIN by changing only one characteristic at a time, i.e., dataset size, object width, variation in object width and translation distance. We exclude the performance measurement when changing the aspect ratio of each object (while keeping the volume of the object constant). As expected we found no significant difference in terms of performance for THERMAL-JOIN and the competing approaches because none of the techniques depend on the aspect ratio of the spatial objects used. Similarly, for the skewed benchmark, we change the spread of the cluster, i.e., the standard deviation of the normal distribution and the number of clusters.

5.3.1 *Dataset Size*

In a first experiment we study the impact of the dataset size by increasing the dataset size of a uniform benchmark from 10 million objects (400MB) to 50 million objects (2GB). As the results in Figure 9(a) show, using more objects in the dataset degrades the performance of the loose Octree further compared to TOUCH and the CR-Tree. In this experiment the THERMAL-JOIN outperforms competing approaches and yields a speedup of 7.1× when compared to the second best technique, the CR-Tree. Some results for TOUCH, the loose Octree and EGO are missing as they did not finish execution within the 72 hour time limit we set.

5.3.2 *Object Size*

We next increase the width of the objects from 5 to 25 units uniformly so that all objects have the same extent in all dimensions. This increases the volume of the objects cubically and therefore increases the selectivity of the join. As Figure 9(b) shows THERMAL-JOIN provides a speedup of 7.2× compared to the CR-Tree.

5.3.3 *Variation in Object Size*

Although the objects in scientific simulations have the same size, in Figure 9(c) we also experiment with different object sizes (widths) of the smallest and largest object to further study the behavior of THERMAL-JOIN. The object width difference of zero in Figure 9(c) denotes that all objects have the same width of 15 units, while the difference of 4 units means the smallest object has a width of 13 units and the largest object 17 units (in all dimensions). Crucially, although the width changes linearly on the x-axis the difference in terms of volume of the objects increases cubically, e.g., in case of a width difference of 16 units, the volume of the largest object is 35 times larger than the smallest object.

Objects are chosen uniform randomly between the two extreme object sizes. In the best case, when all objects have

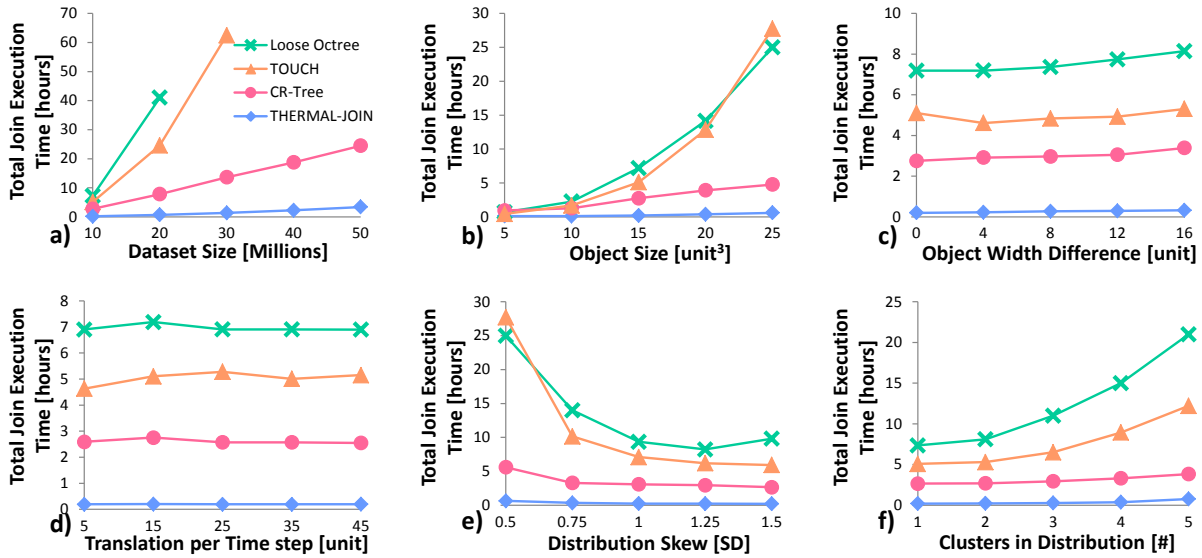

Figure 9: Sensitivity analysis using a synthetic benchmark.

the same size (difference zero), THERMAL-JOIN achieves a speedup of 13.7×. For the worst case the performance of THERMAL-JOIN decreases but it still achieves a speedup of 10.4× over related work. The performance drops because in the worst case the T-Grid needs to be built for each P-Grid as it does not satisfy the condition of a hot spot.

5.3.4 *Temporal Resolution*

In this experiment we change the temporal resolution of the simulation by increasing the distance each object is allowed to move between any two time steps. Doing so does not affect the distribution of the dataset.

The results shown in Figure 9(d) confirm the results expected, i.e., none of the techniques depend on the dynamics of the simulation. While TOUCH, CR-Tree and the loose Octree are rebuilt from scratch at every time step, THERMAL-JOIN uses incremental building and garbage collection to keep the overhead of building the index low, both in terms of space and performance. In the extreme case where an object always changes its P-Grid cell (making big jumps), the overhead is equal to rebuilding the P-Grid index from scratch. This effect is visible when we move the objects by 45 units (and the speedup is reduced by a small fraction from 13.3× to 13.0×). Similarly, the garbage collection strategy loses its effectiveness and the memory footprint increases by 27%.

5.3.5 *Distribution Skew*

To test sensitivity to distribution skew, we use a skewed benchmark and change the spread of the normal distribution to measure the impact on the performance. Using the standard deviation <1, reduces the spread and increases the join selectivity. Figure 9(e) shows higher join selectivity favors THERMAL-JOIN and therefore it outperforms competing approaches and achieves a speedup of 8.8×.

5.3.6 *Clustering*

Another way to vary the spread of the skewed distribution is to create several small clusters rather than just one. Keeping the standard deviation constant at one, we divide the same number of objects among many clusters. The density around the smaller clusters is relatively lower than the density of a single cluster. Although THERMAL-JOIN outperforms competing approaches, the speedup drops from 12× to 5× because the join is no longer as selective as with a single cluster as Figure 9(f) shows.

6. THERMAL-JOIN ANALYSIS

In this section we present experiments that highlight the internal working of THERMAL-JOIN, i.e., how do the join phases and the memory footprint change when we change the P-Grid resolution. We conclude by briefly discussing the applicability of THERMAL-JOIN based on the design choices we made along with limitations of the approach.

6.1 P-Grid resolution

In the following experiment we analyze and break down the performance of THERMAL-JOIN and describe how it changes as we change the grid resolution. In Figure 10(a) we vary the grid resolution r from 0.5 to 2 and then observe the cost of building and performing the self-join. For the experiment we use a real neural workload with one million objects with object width equal to $15 micron^3$. As we increase the resolution from $r > 1$ the cost of internal join starts to become substantial. This is because the P-Grid cells are no longer hot spots. As we decrease the resolution, i.e., $r < 1$ we increase the building and join time substantially because there is a considerable number of cells that require a long time to build and then to perform the external join despite all the objects are in hot spots.

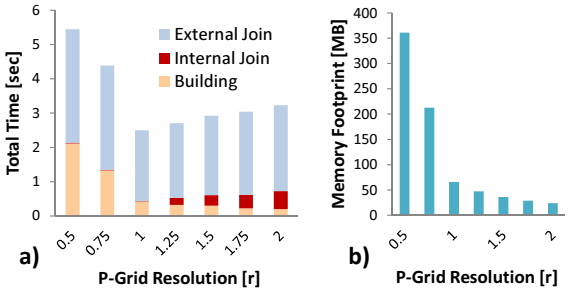

Figure 10: Effect of changing grid resolution r on (a) join time, (b) footprint of THERMAL-JOIN.

The memory required on the other hand only depends on the number of grid cells instantiated and as we increase r so that $r > 1$, fewer cells are needed and thus the footprint becomes insignificant as shown in Figure 10(b).

6.2 Applicability

We do not make any particular assumption regarding the semantics of the simulation application and THERMAL-JOIN thus is applicable to many different problems that require iterative spatial self-join. Obvious non-scientific use cases for the approach are video games. Similar to the problem of scientific simulations, in multi-player games a cut-off radius (region of visibility) is defined for all characters that are changing their location at discrete intervals of time.

Furthermore, the assumption that the number and the shape of objects should remain constant during the simulation does not limit applicability as no part of the THERMAL-JOIN algorithm relies on this assumption.

6.3 Limitations

THERMAL-JOIN is designed to address the challenges of joining highly selective datasets that change unpredictably during the simulation. This means for cases where such extreme access pattern are not observed, simpler solutions can be used. For example, if the join selectivity is very low a straightforward approach may be to use an iterative plane-sweep join. In case the objects are changed over time predictably, trajectory based techniques may be more suitable.

The design choices for THERMAL-JOIN prioritize runtime performance and scalability. In terms of memory footprint, however, spikes are observed due to the iterative tuning. This can be improved by avoiding a very fine resolution grid that would exceed a memory quota given by the user.

7. CONCLUSIONS

In this paper we present the THERMAL-JOIN approach, a high performance and scalable solution for executing spatial self-joins iteratively in main memory. The algorithm is practical to use, i.e., it does not require tuning elaborate configuration parameters and it is resilient to different workload characteristics as we show in Section 5.3. The approach uses the novel concept of spatial hot spots that improve the performance for workloads with high join selectivity such as scientific simulations. Using real neural workloads we show that the approach achieves speedup of 8 to 12× when compared to the state-of-the-art and remains competitive in terms of memory footprint.

Acknowledgements

This work is supported by the Hasler Foundation (Smart World - Databasing the Brain, No 11031) and the EU Framework Programme (FP7/2007-2013) under grant 604102 (HBP).

8. REFERENCES

[1] V. Akcelik et al. High Resolution Forward and Inverse Earthquake Modeling on Terascale Computers. In *Supercomputing '03*.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient Exact Set-Similarity Joins. In *VLDB '06*.

[3] G. Blelloch and G. Narlikar. A Practical Comparison of N-Body Algorithms. In *Parallel Algorithms '97*.

[4] C. Böhm, B. Braunmüller, F. Krebs, and H.-p. Kriegel. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. *SIGMOD Record*, 30, 2001.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD '93*.

[6] S. Chen, C. S. Jensen, and D. Lin. A Benchmark for Evaluating Moving Object Indexes. In *VLDB '08*.

[7] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento. ST2B-tree: A Self-tunable Spatio-temporal B+-tree Index for Moving Objects. In *SIGMOD '08*.

[8] J. Dittrich, L. Blunschi, and M. A. Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD '09*.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 3rd edition, 2000.

[10] D. Frenkel and B. Smit. *Understanding Molecular Simulation*, volume 1. Academic Press, 2001.

[11] R. A. Gingold and J. J. Monaghan. Smoothed Particle Hydrodynamics-theory and Application to Non-spherical Stars. *Astrophysical journal*, 181:375–389, 1977.

[12] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. Mohania. Processing Multi-way Spatial Joins on Map-reduce. EDBT '13.

[13] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD '84*.

[14] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of K-nn and Spatial Join Queries on Continuously Moving Points. *ACM TODS*, 31(2), 2006.

[15] C. L. Jackins and S. L. Tanimoto. Oct-trees and Their Use in Representing Three-dimensional Objects. *Computer Graphics and Image Processing*, 14(3), 1980.

[16] E. H. Jacox and H. Samet. Spatial Join Techniques. *ACM TODS*, 32(1), 2007.

[17] E. H. Jacox and H. Samet. Metric Space Similarity Joins. *ACM TODS*, 33(2), 2008.

[18] K. Kim, S. K. Cha, and K. Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In *SIGMOD '01*.

[19] N. Koudas and K. Sevcik. Size Separation Spatial Join. In *SIGMOD '97*.

[20] D. Kwon et al. An Adaptive Hashing Technique for Indexing Moving Objects. *TKDE*, 56(3):287–303, 2006.

[21] Y. Kwon, D. Nunley, J. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable Clustering Algorithm for N-Body Simulations in a Shared-Nothing Cluster. In *SSDBM '10*.

[22] S. Leutenegger et al. STR: a Simple and Efficient Algorithm for R-Tree Packing. In *ICDE '97*.

[23] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient Processing of K Nearest Neighbor Joins Using MapReduce. In *VLDB '12*.

[24] H. Markram et al. Introducing the Human Brain Project. In *European Future Technologies '11*.

[25] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26:40–49, 2003.

[26] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. TOUCH: In-Memory Spatial Join by Hierarchical Data-Oriented Partitioning. In *SIGMOD '13*.

[27] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD '96*.

[28] J. M. Patel et al. STRIPES: an Efficient Index for Predicted Trajectories. In *SIGMOD '04*.

[29] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 1993.

[30] H. Samet, J. Sankara, and M. Auerbach. Indexing Methods for Moving Object Databases: Games and Other Applications. In *SIGMOD '13*.

[31] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. In *VLDB '14*.

[32] Y. Silva, S. Pearson, and J. Cheney. Database Similarity Join for Metric Spaces. In *Similarity Search and Applications*. 2013.

[33] C. Silvestri et al. GPU-Based Computing of Repeated Range Queries over Moving Objects. In *International Conference on Parallel, Distributed and Network-Based Processing*, 2014.

[34] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An Experimental Analysis of Iterated Spatial Joins in Main Memory. In *VLDB '14*.

[35] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB '03*.

[36] F. Tauheed, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. OCTOPUS: Efficient Query Execution on Dynamic Mesh Datasets. In *ICDE '14*.

[37] D. Šidlauskas and C. S. Jensen. Spatial Joins in Main Memory: Implementation Matters! In *VLDB '15*.

[38] D. Šidlauskas, S. Šaltenis, and C. S. Jensen. Processing of Extreme Moving-object Update and Query Workloads in Main Memory. *VLDB '14*.