4 Parallel Techniques for Collision Detection

Collision detection is a fundamental task in games and virtual environments [Ericson05, Bergen03, Eberly06] and it is usually very resource consuming. The collision detection can be implemented using a two step system. In step one, known as Broad Phase Collision Detection, a high level data structure is used to quickly detect the objects that may be colliding with each other(the set of these objects is known as PCS-Potentially Colliding Set). For example, an octree may be used as such structure, and the objects that reside on a specific node can be considered as potentially colliding. Step two, known as Narrow Phase Collision Detection, is the expensive collision detection between the objects in the Potencially Colliding Set.

This chapter proposes a parallel algorithm for broad-phase collision detection based on hierarchical uniform grids. A parallel implementation for the narrow phase is also proposed.

4.1 Broad Phase Collision Detection with Parallel Hierarchical Grids

There are many methods for broad phase collision detection, which can be found in the excellent survey by Ericson [Ericson05]. Space-partitioning structures have been extensively used to optimize the broad phase. Although hierarchical data structures like octrees, quadtrees or BSPs are more general approaches, uniform-spatial partitioning are more suitable for parallel implementation. The idea (as presented in [Ericson05]) is to divide the space into a uniform grid, such that a grid cell is at least as large as the largest object (Figure 4.1 illustrates a uniform 2D grid). As can be seen in the figure, the objects of this 2D grid are considered to have an axis aligned bounding box (ABB) containing the object. Each cell contains a list of each object whose position is within that cell. In this method, it is necessary to test collision only for the objects that share the same cells. This greatly reduces

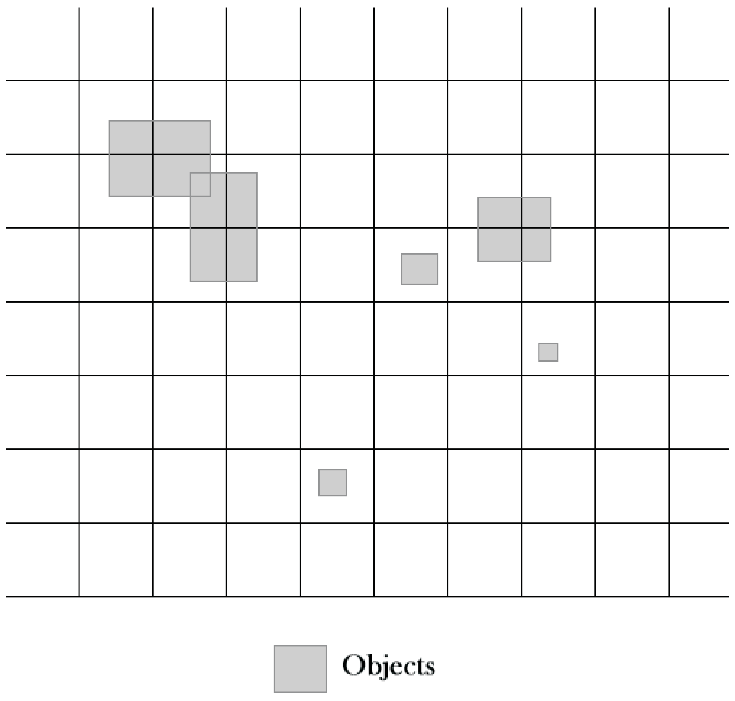


Figure 4.1: 2D Grid dividing the space into equal sized cells.

the number of collision tests that each object must execute. The condition for a cell to be at least as big as the largest object can cause unnecessary computation when there is great disparity in size amongst the total set of objects. Le Grand [LeGrand07] presents a very efficient implementation for a simplified uniform-spatial subdivision with CUDA [Kirk10]. However, Le Grand's implementation does not deal with the above-mentioned problem of disparity in size. This problem can be solved by the method of hierarchical grids firstly proposed by Brian Mirtich in his PhD thesis [Mirtich96], which is also presented in [Mirtich98]. More recent extensions to hierarchical grids [Mathias07, Teschner03] may pose difficulties to parallel algorithms, but this is an issue for further research.

The insertion of an object in the grid is a fast operation. In order to find the location in the grid of a position, we divide each position coordinate by the cell size, as shown below for a 2D grid:

$$\text{grid cell x} = \frac{\text{position x}}{\text{cell size}} \tag{1}$$

$$\text{grid cell y} = \frac{\text{position y}}{\text{cell size}} \tag{2}$$

The main problem with the grid structure is finding the right size for the cell. If the cell size is too small, many cells will need to be updated whenever an

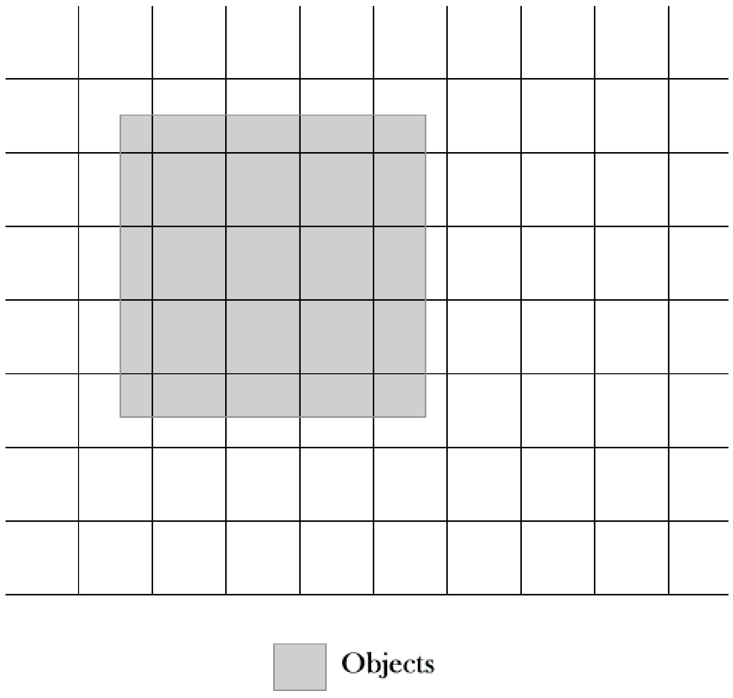


Figure 4.2: A grid with small cell size and big objects.

object moves (figure 4.2) resulting in reduced performance. Another problem is when the cells are too big resulting in many objects occupying the same cell (figure 4.3). The objects occupying the same cell need to be tested against each other resulting in O(n^2) tests. If many objects occupy the same cell, the effectiveness of the grid will be reduced.

If a scene has objects that vary greatly in size, it becomes even harder to find a proper cell size. In order to solve the problem of disparity in size, we can use a hierarchy of grids occupying the same space, where at each level the cell size changes. The objects now are added to the grid that has a cell size big enough to contain the object. This way, the maximum number of cells an object can reside is 4. This happens when the object passes the horizontal and vertical border of his cell. Since the cell has enough space to hold the object, it will occupy at most 4 grid cells.

To test for collision, the objects must now test against objects in the cells not only from their level but from other levels as well. This is because other, bigger objects for example, may occupy the same space as smaller objects, but the bigger objects are in another grid level.

Another problem with grids is the memory requirement they have for big scenes. The memory requirement is a function O($h * w$) where h is the grid height and w is the grid width. As h and w increase in size the memory

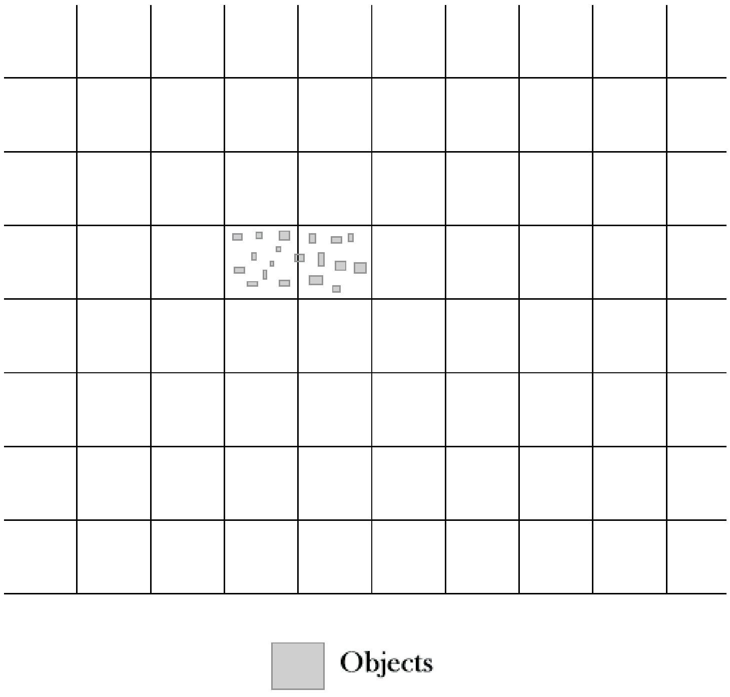


Figure 4.3: A grid with big cell size and small objects.

requirement will become unfeasible.

A solution for this memory problem is to transform the grid into a hash table using the cell position and level as key [Ericson05]. This way, we can have a scene of any size that the memory requirements do not increase. The only drawback of this method is that now we must do a quick test for every two objects in the same cell to see if they are truly close or not. Two objects can reside on the same cell even being far apart, this can happen when their keys created by the hash function collide on the same position.

Our implementation of a hierarchical grid saves the object in a single cell in a 2D grid. This means that we will only save the object reference in a single cell of the 2D grid, even if the object occupies 4 grid cells. The cell that will maintain a reference to the object is selected by the top left corner of an axis aligned bounding box that surrounds the object. Our implementation also uses multiple threads for processing the collision detection.

The pseudocode for adding objects to the grid is presented on algorithm 6.

The algorithm first finds a cell size that can contain the object being added. *maxLevels* is the maximum number of levels that the grid can handle and *cellSize* is a buffer that defines the cell size at each level. The position of the ABB used to find the grid position is the top left of the ABB

Algorithm 6 AddObject(object)

```
1: $greaterSize$ = max(object height, object width)
2: $level = 0$
3: while $level < maxLevels$ and $cellSize[level] < greaterSize$ do
4:     increment level
5: end while
6: $object.level = level$
7: $object.previous = null$
8: $object.bucket = ComputeHashBucketIndex(\frac{object.min.x}{cellSize[level]}, \frac{object.min.y}{cellSize[level]}, level)$
9: if $grid[object.bucket] \neq null$ then
10:     $grid[object.bucket].previous = object$
11: end if
12: $object.next = grid[object.bucket]$
13: $grid[object.bucket] = object$
14: $objects.insert(object)$
```

(min_x and min_y are considered the top left of the ABB). The function *ComputeHashBucketIndex* is called to provide the bucket index in the grid hash table. The pseudocode for the function is presented on algorithm 7.

Algorithm 7 ComputeHashBucketIndex(cellx, celly, cellz)

```
1: $index = (cellx + celly + cellz)\%MAX\_BUCKETS$
2: if $index < 0$ then
3:     negate $index$
4: end if
5: return $index$
```

The *AddObject* function then adds the object to the bucket received. Each bucket stores their objects using a double linked list. This way, adding and removing objects is done in O(1). In the end, the object is also added to a objects buffer used in the proximity tests.

The pseudocode for removing objects is presented on algorithm 8. The algorithm simply redefines the references from the objects before and after the object being removed.

The pseudocode for processing collision detection between the objects is presented in algorithm 9.

The algorithm first divides the workload of testing object proximity by customizing the intervals buffer. The intervals buffer defines the start and end indices of the objects that each processor must handle. *closePairs* is a buffer that stores all object pairs that are potentially colliding. This function clears this buffer making it empty. The algorithm then starts the *DetectProximityThread* that makes use of the intervals buffer to test object

Algorithm 8 RemoveObject(object)

```
1: if $object.previous \neq null$ then
2:     $object.previous.next = object.next$
3: end if
4: if $object.next \neq null$ then
5:     $object.next.previous = object.previous$
6: end if
7: $objects.remove(object)$
```

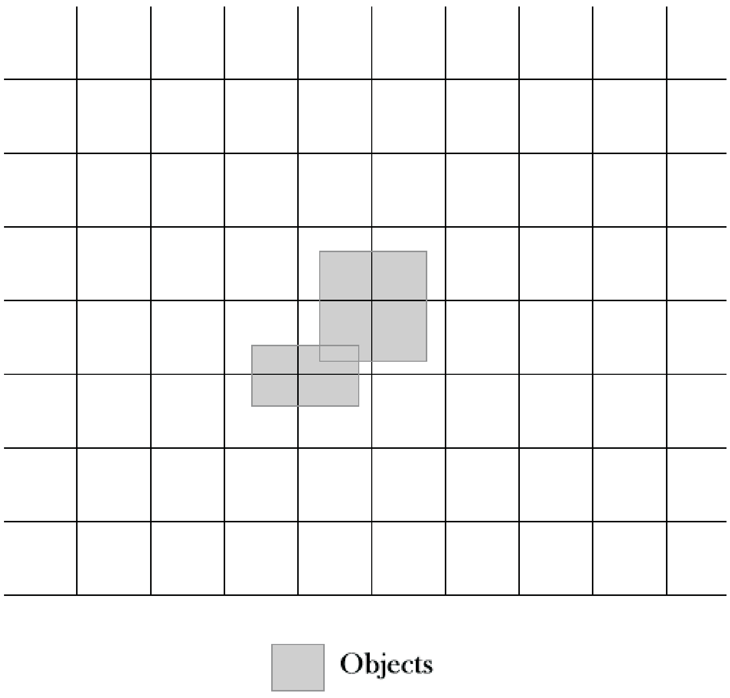


Figure 4.4: An object may be colliding against objects in the southwest cell.

proximity. The *DetectProximityThread* adds pairs to the *closePairs* buffer. After all threads finish, the *TestCollisionThread* is started on all processors. The *TestCollisionThread* calls the proximity callbacks that handle expensive collision detection tests between pairs of objects using the *closePairs* buffer.

The *DetectProximityThread* tests the proximity of a group of objects, the pseudocode for this function can be seen on algorithm 10.

The thread uses the interval from the intervals buffer in order to process its share of objects. Each object is tested not only against objects in its own level but also against objects on higher levels. The lower level cells need not to be tested as objects on those levels will also proceed testing against the higher level cells. Each object must be tested against 5 cells. Four are the cells that the object can intercept, and one is for the case shown in figure 4.4.

Algorithm 9 ProcessCollisionDetection()

```
1: $workSize = \frac{objectCount}{numProcessors}$
2: {divide the workload of detecting proximity between objects}
3: $intervals[0].start = 0$
4: $intervals[0].end = workSize - 1$
5: for each $i$ from 1 to $numProcessors - 2$ do
6:     $intervals[i].start = intervals[i-1].start + workSize$
7:     $intervals[i].end = intervals[i-1].end + workSize$
8: end for
9: $intervals[numProcessors - 1].start = intervals[numProcessors - 2].start + workSize$
10: $intervals[numProcessors - 1].end = objectCount - 1$
11: {detect proximity}
12: $closePairs.clear()$
13: for all processors do
14:     start the DetectProximityThread sending the proper interval as parameter
15: end for
16: Wait until all threads finish their execution
17: for all processors do
18:     start the TestCollisionThread
19: end for
20: Wait until all threads finished
```

The *TestCollisionThread* calls the collision detection callback of all objects that are close to each other. Inside the callback should be implemented the expensive detailed collision detection. The *TestCollisionThread* divides the workload of the expensive tests on each processor. The pseudocode can be seen on algorithm 11.

(a) Analysis

On the *TestProximity* algorithm, the main workload is done in the *DetectProximityThread* and *TestCollisionThread*. The worst case for *TestProximity* is when all objects are occupying the same bucket on the grid. In this case all objects will test proximity against all other objects resulting in O(n^2) proximity and collision tests. This case however is very rare to occur in a game as objects in a scene generally start separated from each other and are pushed back when they intersect.

Algorithm 10 DetectProximityThread(interval)

```
for each $i$ from $interval.start$ to $interval.end$ do
  $obj = objects[i]$
  for $level = obj.level$ ; $level < maxLevels$ ; $level++$ do
    $x1 = \frac{obj.min.x}{cellSize[level]}$
    $y1 = \frac{obj.min.y}{cellSize[level]}$
    $x2 = \frac{obj.max.x}{cellSize[level]}$
    $y2 = \frac{obj.max.y}{cellSize[level]}$
    for $x = x1$ ; $x \leq x2$ ; $x++$ do
      for $y = y1$ ; $y \leq y2$ ; $y++$ do
        $bucket = ComputeHashBucketIndex(x, y, level)$
        for $p = grid[bucket]$ ; $p \neq null$ ; $p = p.next$ do
          if $p$ equals $obj$ then
            continue
          end if
          if IsClose($obj$, $p$) then
            add the pair $obj$ and $p$ to $closePairs$ buffer if the pair is not
            already added
          end if
        end for
      end for
    end for
    {now test the southwest position if the bounding box takes 2 vertical
    cells}
    if $y2 > y1$ then
      $bucket = ComputeHashBucketIndex(x1 - 1, y2, level)$
      for $p = grid[bucket]$ ; $p \neq null$ ; $p = p.next$ do
        if IsClose($obj$, $p$) then
          add the pair $obj$ and $p$ to $closePairs$ buffer if the pair is not
          already added
        end if
      end for
    end if
  end for
end for
```

Algorithm 11 TestCollisionThread(threadBuffer)

```
1: while true do
2:    enter critical section
3:    add to threadBuffer n pairs from closePairs
4:    leave critical section
5:    if threadBuffer is empty then
6:       break loop
7:    end if
8:    for all pairs in threadBuffer do
9:       ProcessDetailedCollisionDetection(pair)
10:   end for
11:   threadBuffer.clear()
12: end while
```

4.2 Narrow Phase Collision Detecting Load Balancing

A multicore architecture involves adequately balancing the load between each core and reducing the synchronization parts in the code. Another area that must receive special attention is effective cache use. Since each core may have its own level 1 cache, it is important that each core does not invalidate the cache lines in other cores by updating on a shared cache line [Grama03]. Failing to do so, may result in poorer performances from the use of multiple cores when compared to the use of a single core. The cache memory on CPUs is composed of several cache lines. Each cache line stores a block of bytes from main memory. When the CPU updates the cache memory, it copies a full block of bytes from main memory to the cache memory (even if it doesn't need the full block).

The foundation of this section comes from the work on parallel particle processing used for physical simulations [Andrews99].

(a) The Algorithm

The collision detection of N objects in a single processor system can be implemented using a pseudocode as shown on algorithm 12.

(b) Analysis

Algorithm 12 receives a vector with the objects that are going to be tested for collision. The first object tests collision against n-1 other nodes. The second object tests against n-2 and this process continues until the last object tests against a single object. The number of iterations of the outer loop is n-1. This results in a number of tests equal to:

Algorithm 12 ProcessCollision(objects)

```
1: for each i from 0 to objects.size()-2 do
2:   for each j from i+1 to objects.size()-1 do
3:     if TestCollision(objects[i], objects[j]) equals true then
4:       objects[i].onCollision(objects[j]);
5:       objects[j].onCollision(objects[i]);
6:     end if
7:   end for
8: end for
```

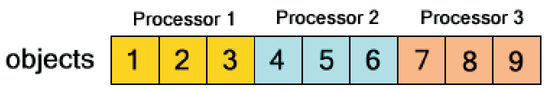


Figure 4.5: Simple Parallel Collision Detection.

$$tests = (n-1) + (n-2) + .. + 1$$

$$2 * tests = ((n-1)+1) + ((n-2)+2) + ... + (1+(n-1))$$

$$tests = \frac{(n-1) * n}{2}$$

The number of tests for algorithm 12 represent an algorithm complexity of O(n^2).

The method *TestCollision* implements the collision detection between two objects. This method usually tests the bounding volumes of the objects for intersection and returns true if an intersection took place. If it is the case, the objects receive collision events receiving as parameter the object they collided with.

A simple way to implement the same collision detection above in a multi processor system is to divide the objects vector in N blocks, one for each processor as shown in figure 4.5. The pseudocode for the approach above is presented on algorithm 13.

processorIdx is an index representing the processor and receives values from zero to the number of processors minus one.

The problem with this approach is that the processing load is unbalanced between the processors. As shown in figure 4.6, processor 1 makes 8 + 7 + 6 = 21 collision tests, while processor 2 makes 5 + 4 + 3 = 12 tests and processor 3 makes 2 + 1 = 3 tests.

A better approach is to make n iterations over the object list where n = (number of objects) / (number of processors). On each iteration, if a processor takes the most expensive object to process, on the next iteration it takes the least expensive. If a processor takes the second most expensive, on the next

Algorithm 13 ProcessCollision(processorIdx, objects)

```
1: processingBlock = $\frac{objects.size()}{\text{number of processors}}$
2: startIdx = processorIdx × processingBlock;
3: if processorIdx == number of processors-1 then
4:   endIdx = objects.size() - 1
5: else
6:   endIdx = (processorIdx + 1) × processingBlock;
7: end if
8: for each i from startIdx to endIdx-1 do
9:   for each j from i+1 to objects.size()-1 do
10:     if TestCollision(objects[i], objects[j]) equals true then
11:       objects[i].OnCollision(objects[j])
12:       objects[j].OnCollision(objects[i])
13:     end if
14:   end for
15: end for
```

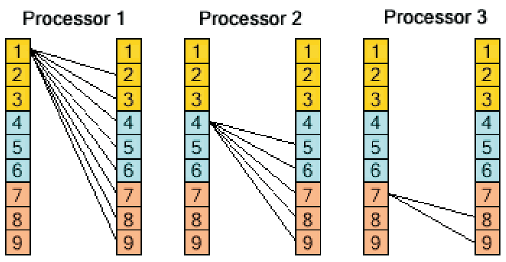


Figure 4.6: The processing of the first object for each processor.

iteration it takes the second least expensive and so on, for all iterations, as shown in figure 4.7.

This manner of object delegation makes processor 1 handle 8 + 3 + 2 = 13 collision tests, processor 2 handles 7 + 4 + 1 = 12 collision tests and processor 3 handles 6 + 5 + 0 = 11 collision tests. The pseudocode that implements this manner of processing is presented on algorithm 14.

By proceeding as defined in the algorithm, the processors constantly change the workload on each cycle. If a processor takes a lot of work in a cycle, it takes less work in the following cycle compared to the other processors. In the end, all processors receive a very balanced amount of work.

(c) Collision Response Results

After detecting a collision between two objects, the collision response processing takes place. An important problem to be solved is how to access the objects and save the results. One solution could be to use a synchronization mechanism such as a mutex on each object [Andrews99]. A mutex is a software

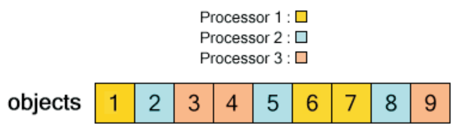


Figure 4.7: A better selection for each processor to handle.

Algorithm 14 ProcessCollision(processorIdx, objects)

```
1: delta = number of processors × 2 - 1
2: for i = processorIdx ; i < objects.size() - 1 ; i += delta - 2 * (i % number
   of processors) do
3:   for each j from i+1 to objects.size()-1 do
4:     if TestCollision(objects[i], objects[j]) equals true then
5:       objects[i].OnCollision(objects[j])
6:       objects[j].OnCollision(objects[i])
7:     end if
8:   end for
9: end for
```

component that allows a programmer to implement mutual exclusion between threads. When a processor needs to compute the collision results on the object, it makes use of that mechanism and saves the results on the object.

The required sychronizations could be considerably reduced by using a different solution. For each object, there could be a buffer where each index is related to a processor, that would receive the collision response results. When calling OnCollision on the objects, the processor index that is to be used for accessing the specific buffer should also be passed as a parameter, as shown in the algorithms 15 and 16.

Algorithm 15 ProcessCollision(processorIdx, objects)

```
1: delta = number of processors × 2 - 1
2: for i = processorIdx ; i < objects.size() - 1 ; i += delta - 2 * (i % number
   of processors) do
3:   for each j from i+1 to objects.size()-1 do
4:     if TestCollision(objects[i], objects[j]) equals true then
5:       objects[i].OnCollision(processorIdx, objects[j])
6:       objects[j].OnCollision(processorIdx, objects[i])
7:     end if
8:   end for
9: end for
```

After all the collision detection and response are computed, the results on each object are merged and the final collision response result is achieved as shown in figure 4.8. At the end of the collision detection and response computing the processors must be synchronized before the merging process

Algorithm 16 OnCollision(processorIdx, object)

```
1: response[processorIdx] = ProcessCollisionResponse(object);
```

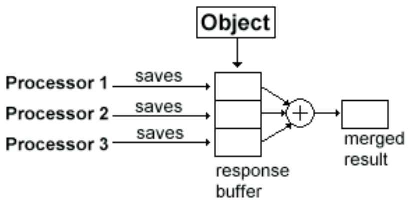
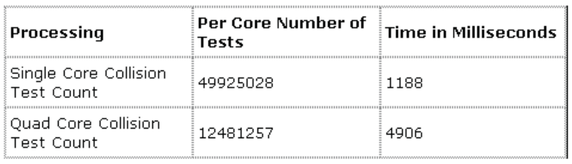


Figure 4.8: Saving the parallel collision response results.



Processing	Per Core Number of Tests	Time in Milliseconds
Single Core Collision Test Count	49925028	1188
Quad Core Collision Test Count	12481257	4906

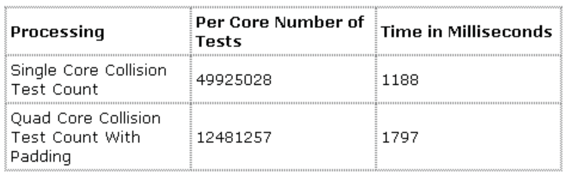
Figure 4.9: Parallel Collision Count Table.

starts on the completed collision results. After the merging is done, the processors must once again be synchronized in order to guarantee that all merging is finished before the processors carry on to other tasks.

(d) Cache Considerations

It is of extreme importance to consider the cache line size when accessing elements of a buffer from multiple processing cores. The reason is that multiple buffer parts may occupy the same cache line, and when a processor alters a position on the buffer it may invalidate the cache of other cores. For example, consider that we would like to count the number of collision tests for each core. A buffer with 4 integer counters consisting of 4 bytes each is created, assuming a quad core system. If the system has a cache line of 64 bytes the whole buffer can reside on a same cache line. Since our Intel Core 2 Computer uses MESI protocol [Intel09] for cache coherency, every time a core increments a counter on a buffer position, the cache line on the other cores is invalidated. When the other cores try to increment their counters they will need to fetch the line again. This can create a performance poorer than a single core execution. The table from figure 4.9 presents the collision test count performance for 10.000 objects on a Intel Core 2 Extreme CPU Q6850 (quad-core) with a clock rate of 3.00 GHz. The first line shows the time it takes to count the number of collision tests using a single core. The second line shows the same test using 4 cores. As can be seen, the parallel approach performs much slower(figure 4.9).

The solution to this problem is to place each counter on different cache



Processing	Per Core Number of Tests	Time in Milliseconds
Single Core Collision Test Count	49925028	1188
Quad Core Collision Test Count With Padding	12481257	1797

Figure 4.10: Parallel Collision Count with Padding Results.

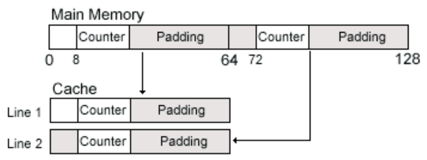


Figure 4.11: Cache line loading of 2 elements of a buffer.

lines. This can be done by changing the size of the buffer elements to the size of the cache line. In order to do this, it is necessary to create a new type that has the counter and some padding bytes to complete a full cache line. Since the tested system has 64 bytes, the created type would have 60 padding bytes as shown in the C code below:

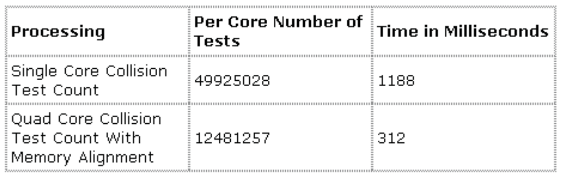
```
struct CollisionCounter
{
    int counter;
    char padding[60];
};
```

The performance after applying padding is presented on figure 4.10:

The problem with applying only padding is that it is not guaranteed that each buffer element will reside in a specific cache line. Cache lines load values in addresses that are multiple of the cache line size. If a processor has a cache line size of 64 bytes, it will load values from addresses 0 to 63, 64 to 127, etc, as shown in figure 4.11.

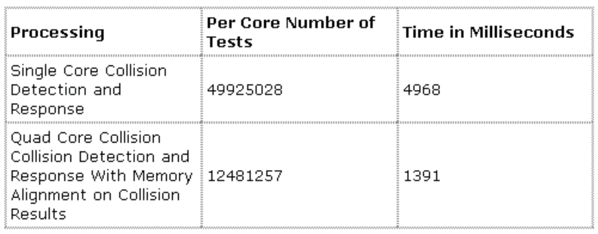
So if a buffer element is not aligned with the cache line size, it will occupy more than one cache line (assuming that each buffer element is set with the size of the cache line). For Microsoft Visual Studio 2008, __declspec(align(cache line size)) can be used to align structures to the cache line. The code is presented below:

```
__declspec(align(64)) struct CollisionCounter
{
    int counter;
};
```



Processing	Per Core Number of Tests	Time in Milliseconds
Single Core Collision Test Count	49925028	1188
Quad Core Collision Test Count With Memory Alignment	12481257	312

Figure 4.12: Parallel Collision Count with Memory Alignment using 10.000 objects.



Processing	Per Core Number of Tests	Time in Milliseconds
Single Core Collision Detection and Response	49925028	4968
Quad Core Collision Collision Detection and Response With Memory Alignment on Collision Results	12481257	1391

Figure 4.13: Parallel Collision Detection with Memory Alignment using 10.000 objects.

It is not necessary to add the padding when using memory alignment since the next element of the counter buffer will also be aligned to the cache line. The performance of the memory aligned collision test count is presented on the table of figure 4.12. As can be seen, the performance is much improved.

In figure 4.13 we present a table with the processing time for computing collision detection and response of 10.000 objects: