Object Oriented Parallelisation of Graph Algorithms using Parallel Iterator

Lama Akeila[1] , **Oliver Sinnen**[2] **and Wafaa Humadi**[3]

The Department of Electrical and Computer Engineering
The University of Auckland, New Zealand.
[1] lake003@aucklanduni.ac.nz, [2] o.sinnen@auckland.ac.nz, [3] whum003@aucklanduni.ac.nz

Abstract

Multi-core machines are becoming widely used which, as a consequence, forces parallel computing to move from research labs to being adopted everywhere. Due to the fact that developing parallel code is a significantly complex process, the main focus of today's research is to design tools which facilitate the process of parallelising code. The Parallel Iterator (PI) is a tool which was developed to automate the process of parallelising loops in OO applications. Graph algorithms can be represented using objects and hence they are excellent use cases for the PI. This paper discusses using the PI to parallelising graph algorithms such as breadth-first search (BFS), depth-first search (DFS) and minimum spanning tree (MST). Using the PI to parallelise such graph algorithms required adding some adaptations to the current concept of the PI to handle certain graph algorithms. The PI facilitates the process of parallelising graph algorithms in a way which keeps the parallel code readable and maintainable while exhibiting speedup. Java was used as the implementation language since it is one of the most commonly used object oriented languages. The parallelised graph algorithms were tested on different graphs and trees with different densities, granularity and structures. The experimental results show that the parallelised graph algorithms exhibit good speedups.

Keywords: parallel computing, object oriented parallelisation, Parallel Iterator, graph algorithms.

1 Introduction

With the introduction of multi-core processors, the importance of parallel computing has accelerated into the mainstream and hence, parallel computing technologies have been adopted everywhere (Reinders 2007). The expected performance speedup gained by increasing the number of processors depends on the problem to be solved and the algorithm which is used (Rajasekaran & Reif 2007). This signifies that the software has to be designed in a way which takes advantage of the increased number of processors and hence parallelising software applications becomes a necessity. The process of parallelising software applications is not straight forward since the developers are forced to deal with parallelisation details such as synchronisation between processors, locking of shared resources, race conditions and so on. As a consequence, the main focus of today's research is to develop tools which facilitate the development of parallel applications. Most software applications rely heavily on iterative computations (N.Giacaman & O.Sinnen 2008) (i.e. computations handled by loops). In OO languages loops are handled by Iterator objects. As a consequence, tools for parallelising loops have very significant advantages when developing parallel applications. An example of such a tool is the PI which has been developed and implemented in the ECE department at the University of Auckland (N.Giacaman & O.Sinnen 2008, Akeila 2008).

The Parallel Iterator provides a thread-safe mechanism to iterate through a collection of elements concurrently by multiple threads and hence it eases the process of loop parallelisation in many OO applications. A graph library is an example of an OO application which plays an important role in various fields. Many applications rely on graph algorithms to solve common problems in computer science, chemistry and business (Buckley & Lewinter 2003). Graph libraries can be well implemented with objects. To maintain high productivity, readability and maintainability, the parallelisation should be done in an OO way. As a consequence, an OO tool such as the PI is powerful in terms of producing an OO parallel version of graph algorithms with a readalbe and maintainable code which exhibits speedup. Graphs are excellent use cases for the PI. Given their special structures, some graph algorithms might need adaptations and improvements to the PI, which is investigated in this paper. These adaptations are encapsulated by the PI (i.e. all the parallelisation details and the new adaptations are implemented internally by the PI) which, as a consequence, requires little or no code restructuring when using the PI to parallelise graph algorithms.

Parallelising three main graph algorithms using the PI is discussd in this paper: Breadth-First Search (BFS), Depth-First Search (DFS) and Minimum Spanning tree (MST). Section 2 introduces the PI. An overview about graph theory is included in section 3. Section 4 discusses the BFS and its parallelised versions while sections 5 and 6 discuss DFS and MST respectively.

2 The Parallel Iterator

Object oriented programming is widely used by software engineers. It allows software designers to develop high quality software solutions that are reusable and easy to implement and maintain (Craig 2001). Most applications heavily rely on iterative computations which are normally encapsulated inside loops (N.Giacaman & O.Sinnen 2008). As a consequence, the main approach in parallelising OO applications is parallelising loops. In object-oriented languages iterative computations such as loops are handled using iterators. Iterators are objects which allow the developer to traverse collections of elements by calling two main methods: 1) *hasNext* which returns a boolean value indicating whether there is any remaining element in the collection. 2) *next* which returns the actual element. However, conflicts occur if multiple threads are accessing the collection of elements concurrently when one element is remaining in the collection and at least two threads call the *hasNext* method simultaneously. Only one thread gets the next element while the other throws an exception. In addition to that, further parallelisation issues need to be taken into account when traversing a collection of elements concurrently such as load balancing and supporting different scheduling mechanism. Due to the insufficiency of using the normal iterator for parallel systems, the need for a mechanism to traverse collections in a thread-safe and parallel appropriate fashion became evident.

The PI concept has been developed and implemented in the ECE department at the University of Auckland. It allows for an efficient parallel traversal of a collection of elements without resulting in any conflicts between the threads which are accessing the collection simultaneously (N.Giacaman & O.Sinnen 2008). Iterations are distributed among the threads according to the specified scheduling policy. Once the thread finishes its allocated iterations, it exits and waits for the other threads to complete their iterations. Such synchronisation between threads is warranted and the program follows sequential semantics.

The PI has two main methods identical to the conventional sequential iterator, *hasNext* and *next*. It follows the typical semantics of an iterator in that the *hasNext* method is always called before calling the *next* method (i.e. it always checks whether there are still elements remaining in the collection before retrieval). The PI supports both random access collections (i.e. elements can be accessed directly in a constant time $O(1)$) and inherently sequential collections (i.e. elements' access time is proportional to the number of elements in the collection $O(n)$). The PI can be used with different scheduling policies and allows for specifying a chunksize as a method parameter. It supports three main scheduling polices as shown below in Figure 1: static (block and cyclic), dynamic and guided scheduling. Figure 1 shows the three different scheduling policies and how the elements are distributed among the threads in each policy with a collection of 9 elements when 3 threads access it simultaneously.

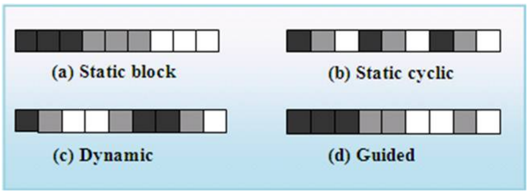


Figure 1: The implemented scheduling policies

The PI simply subsitutes the sequential iterator. Little or no code restructuring is required to be added to the application which is being parallelised by the PI. It also can handle breaks and exceptions and it implements important features such as reductions.

3 Graph Theory

A graph can be defined as a mathematical representation of a relationship or set of relationships between elements (Buckley & Lewinter 2003). Any graph G consists of a nonempty finite list of vertices V and a finite set of edges E that relates the vertices in V. Each edge in the set E consists of two vertices that are related. For example, if $V = \{v_1, v_2, v_3, v_4, ..., v_n\}$ is the total number of vertices in G and $E = \{e_1, e_2, e_3, e_{4,.....,}e_n\}$ is the list of edges in the graph, each edge in E is of the form $\{v_i, v_j\}$ (Buckley & Lewinter 2003). Some graphs have weights $w(v_i, v_j)$ on each edge where w represents the cost of connecting the two vertices, v_i and v_j together.

A path from vertex v_1 to vertex v_4 is the sequence of vertices $\{v_1, v_2, v_3, v_4\}$ which connects vertex v_1 to vertex v_4. If the starting point vertex of the path is the same as the end point vertex, this path is said to produce a cycle and the graph is said to be cyclic graph. If no such cycles occur in the graph it is called acyclic. A graph is called connected if every pair of vertices is connected by a path

Some of the most commonly used graph algorithms are Breadth-First Search (BFS), Depth-First Search (DFS) and Minimum Spanning Tree (MST).

4 Breadth-First search (BFS)

BFS is one of the simplest algorithms for searching graphs and the idea is used by many other algorithms such as Prim's minimum spanning tree algorithm (Cormen, Leiserson, Rivest & Stein 2001). It is also used in other applications in various fields such as image processing (Silvela & Portillo 2001, Cormen et al. 2001). Given a graph G and a source vertex s, the BFS algorithm explores all the vertices reachable from the source vertex s in stages (i.e. the algorithm discovers the vertices reachable from s at distance k before discovering the vertices reachable from s at distance $(k + 1)$ (Cormen et al. 2001). A vertex in level k indicates that the distance from that vertex to the root is k (Buckley & Lewinter 2003). Figure 2 illustrates the different BFS levels of an example tree.

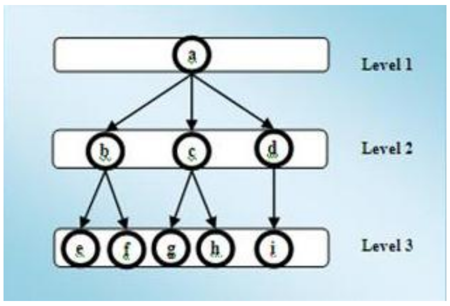


Figure 2: Breadth-First Search Levels

The next section presents the sequential algorithm of BFS and some previous work which was done to parallelise BFS followed by the various approaches which were taken to parallelise the graph algorithm using the PI. The proposed parallel approaches work for any graph. One of those parallel approaches uses the PI directly by passing each level of the tree at a time directly to the PI until no more levels exist. The other parallel approaches are implemented as an extension to the PI concept where the parallel iterator's *hasNext* and *next* methods return the nodes on the fly to the calling processors in a breadth-first order.

4.1 Sequential BFS

The sequential implementation of the BFS relies on processing each level of the tree at a time. It can be implemented using one queue. The following algorithm illustrates the concept where *successors* is a queue which stores the nodes of the tree or graph. Initially the *successors* queue contains the starting vertex of the graph (i.e. the root in the tree case).

Algorithm 1 Sequential BFS.

```
1: while(successors is not empty){
2:   for (every node n in successors){
3:     process n if it is not processed
4:     add successors of n to successor queue end
5:   }
6: }
7: exit
```

As illustrated in Algorithm 1 , the algorithm exits when the *successors* queue is empty (i.e. all nodes have been processed). Processing the nodes level by level ensures the breadth-first order. When BFS is run on trees, node n which is retrieved in line 3 is always processed since every node in a tree has one parent. However in graphs, a node can have more than one parent hence line 3 only processes the node if it was not processed previously.

4.2 Previous Parallel BFS Approaches

Most existing parallel BFS algorithms rely on processing nodes level by level (Yooy, Chowx, Hendersony, McLendonz, Hendrickson & Catalyurek 2005, Zhang & Hansen 2006, Rajasekaran & Reif 2007). All the nodes at a given level can be processed in parallel. A level is usually represented by a certain data structures such as *Queue* which is in turn accessed simultaneously by the different processors (Rajasekaran & Reif 2007). Every time a read or write operations are performed, the queue has to be locked and unlocked to ensure thread-safe behaviour (Rajasekaran & Reif 2007). The parallel algorithm exits when there are no more levels to process and all nodes have been visited.

4.3 Parallelising BFS with PI

In this section, three main approaches to parallelise the BFS algorithm are discussed. The first two approaches are extensions to the current PI as they provide the *hasNext* and *next* method with mechanisms to internally retrieve the next node on the fly in a BFS order. In other words, All the parallelisation details such as synchronizations and communication between threads are encapsulated internally by the iterator's *hasNext* and *next* methods. The third approach uses the conventional PI which was discussed in section 2 directly to parallelise the levels of the tree in a BFS fashion. The first approach uses one concurrent queue, the second uses two sequential queues with a locking mechanism to make the approach thread-safe and the third uses two concurrent queues with the parallel iterator as discussed below.

4.3.1 BFS with One Concurrent Queue

This approach uses one concurrent queue which contains the nodes to be processed. The concurrent queue is a thread-safe Java implementation of *Queue* where all the queuing operations are performed atomically (Sun n.d.) . Each node in this approach is associated with a level attribute which indicates the level at which the node is in the tree as shown in Figure 3.

Figure 3 illustrates when this approach is run on the tree shown in figure 2. The level attributes indicate which level each node is in the tree. Threads retrieve nodes to be processed from the head of the queue and add the successors of the retrieved nodes to the end of the queue as show in Figure 3. Nodes

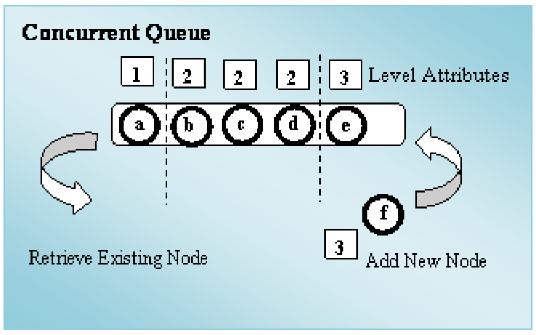


Figure 3: BFS with One Queue

are processed according to their levels in an ascending order. For example , nodes with a level attribute 3 are not processed until all the nodes with a level attribute 2 are processed. To ensure this processing order a global variable, *currentLevel* is shared between all the threads which indicates the current tree level which is currently being processed. It is initialized to 1 (i.e the root level). When all the nodes with a current level has been processed, the global variable *currentLevel* gets incremented to the value of next level. When threads get nodes with a level attribute which is greater than the *currentLevel*, those threads wait until *currentLevel* gets incremented. Algorithm 2 demonstrates the approach which is implemented by the extended PI's *hasNext* and *next* methods. *queue* represents the concurrent queue used in this approach, *id* is the id of each thread which varies from 0 to n where n is the number of processors, *levelsArray* is a 2D array which stores which level is each thread up to and *UpdateCurrentLevel* is a function which updates the value of the global variable *currentLevel* to the minimum positive value stored by *levelsArray*. Every time a thread update its value in *levelsArray*, the function *UpdateCurrentLevel* is called to update the value of the global variable *currentLevel*.

Algorithm 2 Parallel BFS with 1 Queue.

```
1: while (queue is not empty) {
2:   get node d from head of queue
3:   get level attribute , v of node d
4:   add the successors of d to end of queue
5:   levelsArray[id] = v
6:   UpdateCurrentLevel
7:   while(v > currentLevel){
8:     wait until currentLevel gets
       incremented
9:   }
10:  process d
11:}
12:levelsArray[id] = −1
13:UpdateCurrentLevel
14:exit and wait for other threads
   to exit
```

Algorithm 2 calles the *UpdateCurrentLevel* function in lines 6 and 13. This function updates the value of the *currentLevel* by inspecting the level values stored in the *levelsArray* by each thread then setting the global *currentLevel* variable to the minimum positive value. Every time a thread retrieves a different node, the thread records the level value of that node and stores it in the *levelsArray* as shown in Algorithm 2 line 5. When a thread finishes (i.e. line 1 returned false), its recorded level value in the 2D array is set to -1 as shown in line 12 and is ignored by the *UpdateCurrentLevel* function and hence the function will update *currentLevel* to the next positive minimum value when the function gets invoked in line 13. This ensures that the nodes which are stored in the concurrent queue are processed in a level by level manner (i.e. in a breadth-first order).

4.3.2 BFS with Two Queues and Locking

This approach uses sequential queues (i.e. queues which do not support concurrency) with a locking mechanism to enforce a thread-safe access of elements in the queues. One queue in this approach is processed at a time. Each level of the tree is stored in one queue, say queue 1 which gets processed in parallel while the other queue, queue 2 gets populated with the nodes of the next level. The approach starts by processing the root node from queue 1 and writing the next level (i.e. successors) to queue 2. In the following iteration it processes nodes from queue 2 and stores successors in queue 1. This approach continues to alternate between the two queues until all the nodes have been traversed (i.e. both queues are empty). Since the queues which are used in this approach are not concurrent, threads lock the queues before performing read or write operations then unlock the queues when done with the operations. This approach supports specifiying a certain chunksize as a parameter (i.e. number of nodes to be retrieved at once by the thread accessing the queue). Algorithm 3 illustrates the approach which is implemented by the extended PI. *readQ* is the queue which is to be read from, *writeQ* is the queue which is to be written to, *n* is the chunksize and *SwapIfEmpty* is an atomic function which swaps the two queues if the *readQ* is empty.

Algorithm 3 Parallel BFS with 2 Queues and Locks.

```
while(true) {
   SwapIfEmpty
   lock readQ
   if (readQ is not empty) {
      get top n nodes from readQ
      unlock readQ
      lock writeQ
      add successors of n to writeQ
      unlock writeQ
      process the n nodes
   } else {
      unlock readQ
      exit and wait for all other
      threads to exit
   }
}
```

4.3.3 BFS with Two Queues and Parallel Iterator

This approach applies the same concept which was discussed in section 4.3.2 , however it uses 2 concurrent queues and the PI which was discussed in section 2. Each level of the BFS is passed to the PI to be processed concurrently by the threads until all nodes have been processed. The approach is illustrated in Algorithm 4.

Algorithm 4 Parallel BFS with the Parallel Iterator.

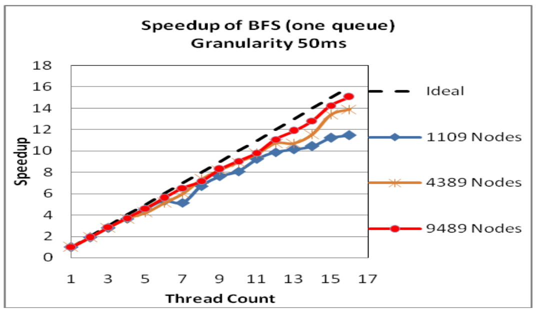
```
1: while(successors is not empty) {
2:   nextLevel = {}
3:   PI = getIterator(successors)
4:   while (PI.hasNext){
5:     node n = PI.next
5:     process n
6:   }
5:   successors = nextLevel
6: }
7: exit
```

Algorithm 4 illustrates how the structure of the parallel BFS approach is similar to any sequential approach. The PI encapsulates the parallelisation details such as synchronization between threads and locking, hence little or no code restructuring is required to parallelise the sequential algorithm. The PI implements a barrier at the end of the iterations loop, hence it is guaranteed that any working thread gets to line 5 of Algorithm 4 when all the other threads have finished executing their iterations (i.e. all nodes in *successors* are processed). Since the PI supports different scheduling schemes and chunksizes, using it to parallelise BFS allows for testing this approach with the different implemented schedule policies to determine which schedule produces the best performance. Performance results are discussed in section 4.4.

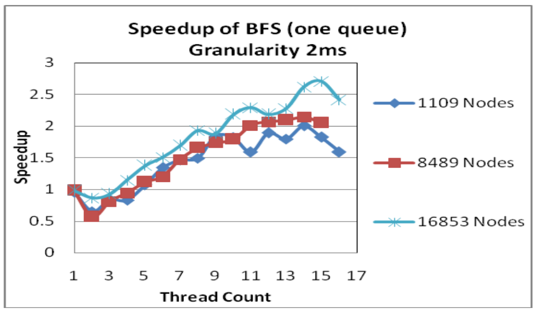
4.4 Parallel BFS Performance

The performance of the three different approaches which were discussed in section 4.3 was evaluated and compared with a large set of experiments using a 16-core machine. The experiments were run on wide trees with different number of nodes and granularity. The granularity was varied by changing the work per node. The speedup of the parallel algorithm is determinded by $\frac{SequentialTime}{ParallelTime}$, where $SequentialTime$ is the time taken to run the BFS by the sequential algorithm discussed in section 4.1 and $ParallelTime$ is the time taken to run the parallel BFS algorithm.

Figure 4 shows the speedup results of the one queue approach discussed in section 4.3.1 with granularity 50 ms and 2 ms per node. The x-axis represents the number of processors (threads) which run the algorithm and the y-axis represents the speedup. A value of 1 on the x-axis represents the parallel code run on 1 processor (i.e. using 1 thread).



(a)



(b)

Figure 4: Speedup Results of BFS with One Queue.

The dashed line shown in Figure 4a represents the ideal expected speedup. Figure 4 shows that the performance of a coarse-grained parallel BFS is better than the fine-grained one as the speedup lines when the granularity is 50 ms per node are closer to the ideal speedup unlike the speedup lines when the granularity is 2ms which is shown in Figure 4b . Figure

4 also shows that the speedup of the parallel algorithm gets better when the number of node increases. However, the parallel BFS with one queue approach produces fluctuating speedup lines when tested on fine-grained graphs as shown in Figure 4b. This is due to the poor termination detection of the implemented algorithm which causes some threads to exit the algorithm early while some other threads are still writing to the concurrent queue.

The speedup results of the parallel BFS approach with two queues and locking discussed in section 4.3.2 are shown in Figure 5. Figure 5a shows the approach when was tested with different chunksizes on a wide graph with 16,853 nodes and a granularity of 2 ms per node. It shows how chunk size 3 produces a slightly better speedup than chunk size 1 since increasing the chunksize reduces the communication overhead between processors due to the locking and unlocking of the queues. However, increasing the chunksize to 8 produces worse performance.

The approach was also tested on wide trees with a varying number of nodes and granularity. Since chunksize 3 produced good performance, the approach was tested with chunksize 3. Figures 5b and 5c show some speedup results.

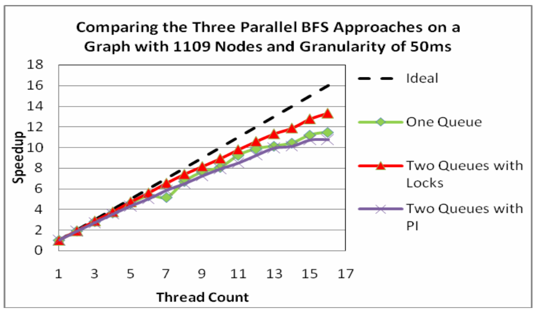
Figures 5b and 5c show how the performance of the parallel BFS gets better as the granularity gets coarser since the speedup lines in Figure 5b are the closest to the ideal speedup. The figure also illustrates that increasing the problem size (i.e. more nodes) enhances the performance of the algorithm. When comparing the 2 ms granularity speedup shown in Figure 5c of this approach with the one queue approach shown in Figure 4b, we find that the fluctuations in the speedup line shown in the one queue approach are not present in this BFS approach hence, the parallel BFS with two queues and locking performs better than the one queue approach when the granularity is small.

The speedup results of the parallel BFS approach with two queues and PI which was discussed in section 4.3.3 are shown in Figure 6. It was tested with a static schedule, dynamic schedule with chunksize 1, 3 and 8 on a wide graph with 16,853 nodes and a granularity of 2 ms as shown in Figure 6a.

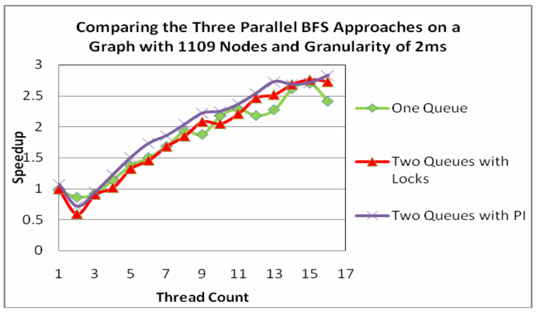
Figure 6a shows that the static schedule produces the worst performance while the dynamic schedule with chunksize 3 produces the best performance which is even better than the results which were produced by the BFS approach with locking shown in Figure 5a.

Similar to the previous approaches, the performance of the parallel BFS which is parallelised by the PI gets better as the granularity gets coarser and increasing the number of nodes enhances the performance of the algorithm. Figure 7 shows the three approaches when tested on a coarse-grained graph (50 ms per node) shown in 7a and a fine-grained graph (2 ms per node) shown in 7b with 1109 nodes.

Figure 7 shows that the parallel BFS with two queues and locks produces a slightly better performance on coarse-grained graphs than the other two approaches. However, the parallel BFS with two queues and PI produces the best performance on fine-grained graphs. For example, the speedup of a fine-grained wide graph with 16,853 nodes using the 2 queues approach with the PI shown in Figure 7b reaches around 2.8 whereas in the one queue approach and the two queues approach with locking, the speedup reaches around 2.4 and 2.7 respectively.



(a)



(b)

Figure 7: Comparing the three parallel BFS approaches on fine-grained graphs.

5 Depth-First Search (DFS)

The depth first search is another searching algorithm which searches deeper in the graph (Buckley & Lewinter 2003). Given a graph G and a source vertex s, nodes are discovered as far as possible along each branch of G before backtracking and discovering the rest of the unvisited edges(Buckley & Lewinter 2003, Cormen et al. 2001). DFS has many applications such as finding strongly connected components, topological sort algorithms and solving puzzles (Cormen et al. 2001, Grama, Gupta, Karypis & Kumar 2003). The order in which nodes are expanded in DFS is illustrated in Figure 8.

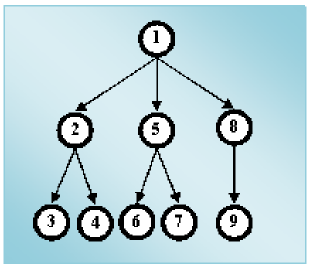


Figure 8: The order in which nodes are expanded in DFS.

The next section discusses the sequential implementation of DFS, some previous approaches which were taken to parallelise the algorithm followed by the various approaches which were taken to parallelise the graph algorithm with the PI.

The sequential DFS is different to the parallel DFS in that a leaf node such as node 3 or 4 in Figure 8 is always discovered before node 5 in the sequential DFS, however the parallel DFS allows node 5 to be discovered before node 4 since multiple threads process the different sub-trees concurrently. In other words, the parallel DFS returns the nodes in topological order.

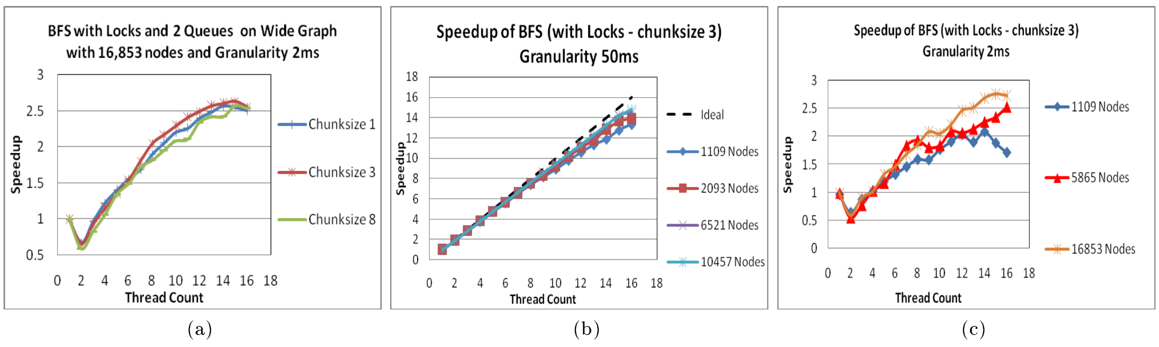


Figure 5: Speedup Results of BFS with two Queues and Locks.

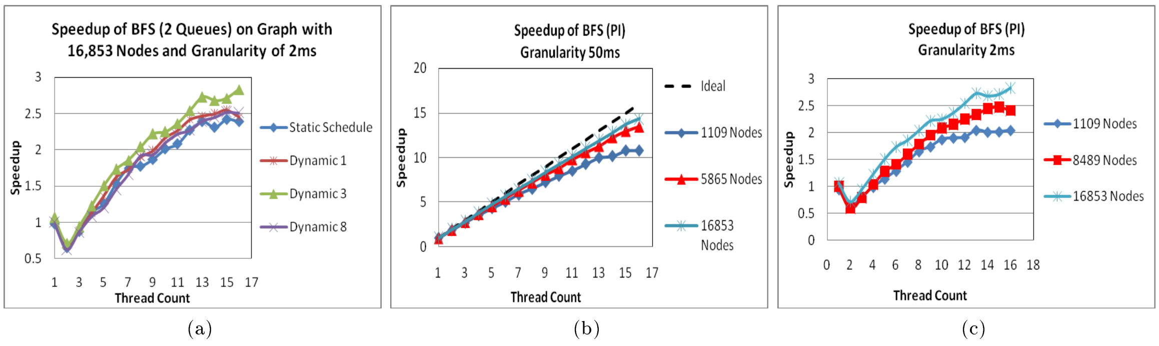


Figure 6: Speedup Results of BFS with Parallel Iterator.

5.1 Sequential DFS

The sequential algorithm uses a stack which is a Last-In-First-Out (LIFO) data structure. The element which is at the top of the stack is processed first. The stack initially contains the root node. As the algorithm proceeds, successors are added to the top of the stack. The nodes in the stack are processed in a LIFO manner until there are no more nodes to be processed. Algorithm 5 illustrates the sequential DFS where *stack* is the LIFO data structure .

Algorithm 5 Sequential DFS.

```
1: while(stack is not empty) {
2:   Get node n from top of stack
3:   Push successors of n
     to stack
4:   process n
5: }
6: exit
```

As illustrated by Algorithm 5 , the algorithm exits when the *stack* becomes empty (i.e. all nodes are processed). Processing the nodes in this fashion enforces a depth-first order.

5.2 Previous Parallel DFS Approaches

Many research attempts have approached the problem of developing a parallel DFS algorithm. According to research studies, the most critical characteristic which determines the performance of the DFS is load balancing (i.e. the mechanism of splitting work between the different processors) (Grama et al. 2003). Several researchers have parallelised DFS by dividing the search space into sub-trees (Reinefeld & Schnecke 1994.). Those individual sub-trees are distributed among the processors to search them in depth-first fashion. To balance the load, a processor which has finished its work attempts to get unprocessed sub-tree form another processor (Reinefeld & Schnecke 1994.). This is called work-stealing. In order to keep all processors busy all the time, a dynamical stack-splitting method can be used (Reinefeld & Schnecke 1994.). In such method, every processor works on the node in its own local stack. When a stack is empty, the processor requests work from other processors. The donor processor splits its local stack to donate unprocessed sub-trees to the processor which requested work (Reinefeld & Schnecke 1994.). Determining the donor processor (i.e. the target processors which will donate unprocessed nodes to the idle processor) can be done by several load-balancing schemes such as Asynchronous Round Robin (ARR), Global Round Robin (GRR) or Random Polling (RP) (Grama et al. 2003). In the ARR scheme, each processor stores an independent variable, *target*, locally which represents the donor processor. It gets incremented by each processor each time work is requested. In the GRR scheme, the value of *target* is stored globally and shared between all the processors. The RP scheme is the simplest since it selects a donor processor randomly every time a processor gets idle. The use of a Last In First Out (LIFO) data structure such as *Stack* ensures that the nodes are processed in a depth-first order.

5.3 Parallelising DFS with PI

The concept of the PI was enhanced to support the implementation DFS. The iterator was extended with a *hasNext* method which returns elements to the calling threads in the DFS, i.e. topological, order on the fly. All the parallelisation details such as synchronizations and communication between threads are encapsulated internally by the iterator's *hasNext* and *next* methods. Two main parallel approaches were implemented, one which uses one global stack and the other uses a local stack for each thread and one global stack shared by all threads. The load-balancing is achieved by the first approach via sharing the nodes (i.e. sub-trees) which are stored in the global stack dynamically. In the second approach, the load-balancing is achieved in two ways: 1) by sharing the node which are stored in the global stack similar to the first approach. 2) by explicitly stealing work from the local stacks of the other working threads when both the global stack and the thread's local stack are empty. The target donor to steal work from is determined randomly as explained in the Random Polling (RP) scheme in section 5.2. The two approaches are explained in the following sections.

5.3.1 Parallel DFS with One Stack Approach

This approach uses one LIFO data structure, a concurrent stack. Threads poll the stack to get a node to process then push successor to the top of the stack. The stack stores nodes which are yet to be processed. The nodes which are stored in the stack are processed dynamically. Once a thread gets idle, it retrieves a new node from the stack. The use of a stack ensures that the nodes are processed in a topological order. Algorithm 6 illustrates the approach where *stack* is the concurrent global stack which is shared between the threads.

Algorithm 6 Parallel DFS with One Stack.

```
1: while(stack is not empty) {
2:   Get node n from top of stack
3:   Push successors of n to stack
4:   process n
5: }
6: exit and wait for all other
   threads to exit
```

Algorithm 6 shows that no explicit locking is performed in this approach since a concurrent stack implementation is used. However, all the locking and unlocking actions are implemented internally by the concurrent stack every time a read or write requests are performed.

5.3.2 Parallel DFS with Local Stacks and Work Stealing

This approach uses one global stack which is shared by all the threads and n local stacks where n is the number of threads. Each thread stores the nodes of its sub-tree in its local stack. Initially the global stack contains the root of the tree and only one thread gets access to it while the other threads are waiting to be woken up by the working thread. The working thread retrieves and processes the root from the global stack, adds one successor to its local stack then pushes the rest of the successors, if they exist, to the global stack to be processed by other threads and wakes up all the waiting threads. Storing only part of successors in the local stack is better than storing all successors at once since it produces better load balancing between threads. It avoids the case where big sub-trees get stored in the local stack of one thread. From this point on, the thread reads from and writes to its local stack until it becomes empty. All the other threads follow the same approach (i.e. when threads get a node from the global stack they push one successor to their local stack and the rest to the global stack then start accessing their local stack until it becomes empty). When the local stack of a thread is empty, the thread retrieves a new node from the global stack. If the global stack is empty, the idle thread tries to get unprocessed nodes from the bottom of a neighboring local stack (i.e. the oldest nodes) using a random polling scheme and stores them in its local stack. If no extra work is available in the neighboring local stacks, the thread exits and waits for the other threads to exit.

5.4 Parallel DFS Performance

The performance of the two different approaches was evaluated and compared with a large set of experiments using the same 16-core machine as before. The experiments were run on wide trees with different number of nodes and granularity.

Experimental results show that the performance of the two approaches is similar when tested on coarse-grained graphs. Figure 9 shows the speedup results of the approachs when tested on a graph with 5003 nodes with a granularity of 40 ms. increasing the granularity and number of nodes produces better performance.

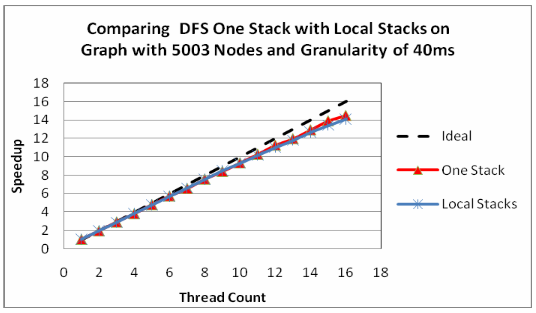


Figure 9: Comparing the two parallel DFS approaches on coarse-grained graph.

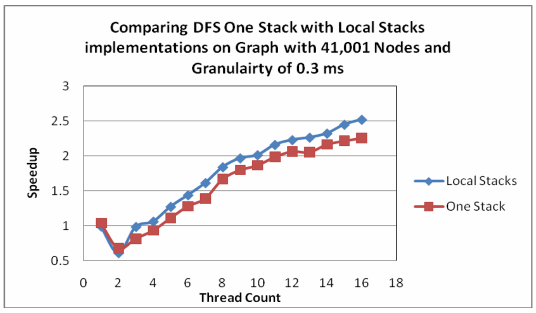


Figure 10: Comparing the 2 parallel DFS approaches on fine-grained graph.

Figure 10 shows a comparison between the two approaches when tested on a fine-grained graphs with 41,001 nodes. The result shows that the local stacks approach performs better than the one stack approach when the granularity is small. This is due to the fact that the overhead produced by the locking and unlocking actions in the local stacks approach is less than the overhead present in the one stack. In the local stacks approach, locking is performed only when threads finish all the nodes in their local stack and attempts to get more nodes from the global stack whereas in the one stack approach locking is always performed when getting nodes since all threads share one global stack.

6 Minimum Spanning Tree (MST)

The Minimum Spanning Tree (MST) is one of the most studied algorithms which has many practical applications in wireless communication, distributed networks (S.Meguerdichian, Koushanfar, Potkonjak & Srivastava 2001) and medical imaging (An, Xiang

& Chavez 2000.). The MST problem determines the minimum-weight path which connects all of the vertices of a given graph G without producing any cycles. The sum of the weights of edges in the path should be the smallest sum possible in the graph G (Cormen et al. 2001). Figure 11 shows the final MST path (in gray) when the algorithm is run on an undirected weighted graph.

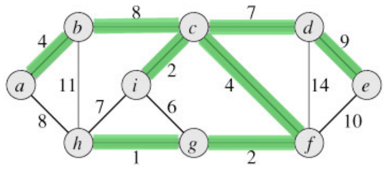


Figure 11: Final MST Path.

6.1 Sequential MST

One of the main algorithms for computing the MST is called Prim's algorithm. It contains a set A that forms the resultant minimum spanning tree. All the edges added to the set A always form a single tree. The safe edge added to A is described as the edge with the minimum weight that is connecting the tree in A to a vertex that is not in the tree (Cormen et al. 2001). The algorithm starts from an arbitrary starting vertex and grows by adding safe edges to the set A until the tree spans all vertices. When the algorithm terminates, set A contains all safe edges that form the minimum spanning tree of graph G. The sequential MST is illustrated in Algorithm 7 where n is the number of vertices in G.

Algorithm 7 Sequential MST.

```
1: Select a vertex v and let V(T)={v}
   and E(T)= {}
2: while(true){
2:    Iterator through edges of v
      and determine the edge e={v,w} with
      minimum weight which connects v
      to another vertex w where w is
      not in V(T)
3:    Add w to V(T) and e to E(T)
4:    If(size of E(T) = n − 1)
5:       exit algorithm
6:    else
7:       v = w
8:    }
9: }
```

6.2 Previous Parallel MST Approaches

One of the solutions to parallelise the MST algorithm is the fast parallel implementation developed by Bader and Cong (D.Bader & Cong 2004.). It allows every processor to start from a different starting vertex and run Prim's algorithm simultaneously (D.Bader & Cong 2004.). In this approach vertices are coloured by the colour of the processor which they were visited by. Every time an edge is to be added to the MST by a particular processor, the processor checks whether the vertex is coloured by its own colour otherwise a collision with another processor occurs. In the case of collisions, the processor stops growing the current sub-tree and exits otherwise it continues until all vertices are visited (D.Bader & Cong 2004.). The final sub trees produced in parallel are merged sequentially in the end to produce the final MST.

Another solution to parallelise the MST algorithm with an adjacency-matrix is to partition the adjacency matrix among the p processors as shown in Figure 12 (Grama et al. 2003).

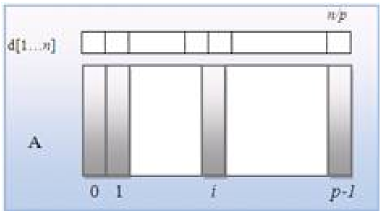


Figure 12: The partitioning of the adjacency matrix and the distance array d .

An array d represents the distance array of length n, where n is the number of vertices in the graph G. Every vertex in the graph has an entry in the distance array d. This entry holds the weight value of the most discovered minimum edge which connects the vertex to another vertex in the graph (Grama et al. 2003). Initially the array entries are initialized to ∞. As a new vertex is discovered and the minimum edge added to the MST, the distance array d is updated with the weights of the incident edges of the newly added vertex. If the weight of one of those edges is smaller than the weight recorded in the array, the entry value is updated to store the edge with the minimum weight. Figure 12 shows how the distance array and the adjacency matrix are partitioned between p processors.

The set of vertices V is partitioned to subsets with $\frac{n}{p}$ consecutive vertices subsets and each subset V_i is assigned to each processor P_i (Grama et al. 2003). Determining the minimum edge and updating the distance array with the newly added vertex are done in parallel. Each P_i stores the part of the distance array d which includes its set of assigned vertices V_i as shown in Figure 12. Each processor computes the edge with the minimum weight from its part of the adjacency list and then the global minimum is computed and stored in process P_0 using an all-to-one reduction process (Grama et al. 2003). The new vertex that is now stored in P_0 is broadcasted to all other processes using one-to-all broadcast, the new edge is added and then each processor updates its relevant distance array portion with the incident edges of the newly added vertex.

6.3 Parallelising MST with PI

The two approaches taken to parallelise MST are based on the approaches discussed in section 6.2. In the first approach processors start from a different starting vertex of the graph and run the MST algorithm simultaneously. The different produced subtrees are merged in the end sequentially. A book keeping module, which keeps track of which nodes have been traversed by which thread, was integrated with the PI to implement this approach successfully.

The second approach uses a distance array and reductions as discussed in section 6.2 to produce a parallel version of MST. This approach uses the PI directly by passing the distance array to the PI to get updated in parallel and using the PI reductions feature to extract the minimum edge.

6.3.1 Parallel MST with Book-Keeping Module

In this approach, a list of the graph vertices is passed to the extended PI. The extended PI issues only the starting points (unvisited nodes) to threads by the *hasNext* and *next* methods. The book-keeping module has two main functionalities; it provides an atomic mechanism which ensures that each node in the graph is processed by only one thread and updates the PI with the status of the nodes (i.e. visited or not) during the run time of the MST algorithm as it keeps a record of which nodes are processed by which threads (i.e. colouring mechanism). Every time a vertex is issued, the book-keeping module colours the vertex by the colour of the processor which visited it. From there, threads continue traversing the graph independently by adding safe-edges to the final MST until either the final MST becomes complete or collisions occur between threads (i.e. more than one thread attempts to access the same node). Collisions are detected by the book-keeping module when more than one thread attempt to access a node. When collisions occur, one thread gets the unprocessed node while the others go back to the PI to get a new unvisited starting vertices. The *hasNext* method of the PI always calls the book-keeping atomic methods to ensure that the issued nodes are unvisited. When all nodes are visited, all the sub-trees which are produced by the different threads in parallel are then merged sequentially.

6.3.2 Parallel MST with Distance Array and Reductions

This approach uses the PI presented in section 2, a distance array d and reductions to produce a parallel version of MST. The distance array d has an entry for every vertex of the graph as explained in section 6.2. Each entry in d stores the edge with the minimum weight encountered so far by the algorithm which connects the corresponding vertex (i.e. the key of array d) with some other vertex. Initially all the entries are set to null. In this approach one edge is added to the final MST edges list $E(T)$ at a time after doing two tasks in parallel in every iteration of the algorithm. The first task is updating d after adding a new vertex to the set of visited vertices $V(T)$ and the second is extracting the edge with the minimum weight from d. At the start of the algorithm a starting vertex is assigned and added to $V(T)$. In every iteration of the algorithm, an edge with the minimum weight which connects a visited vertex in $V(T)$ to an unvisited vertex which is not in $V(T)$ is added to $E(T)$. The unvisited vertex is then added to $V(T)$.

Since two tasks are done in parallel, two instances of the PI are created. One PI instance is used to update d in parallel with the weights of the incident edges of the last vertex v which was added to $V(T)$. This PI instance is initialized with the list of the incident edges v. The second PI is used to extract the minimum edge from d in parallel and is initialized with a list of all vertices which are used as keys in d. When extracting the minimum edge, every thread stores the edge with the minimum weight from its corresponding part of d then the global minimum edge is determined using the PI reductions. The algorithm terminates when the number of added edges is equal to $n - 1$ where n is the number of vertices as shown in Algorithm 8.

Lines 4 and 11 of Algorithm 8 are executed in parallel using the PI. In line 18 the reduction feature implemented by the PI is used to retrieve the global minimum edge. Reductions in the PI are performed in an OO fashion using a *Reducible* object shown in line 10 which manages the different copies of the variable to be reduced(Giacaman, Sinnen & Akeila 2008). PI implements reductions in a way which allows the user to customize the type of reduction to be used which makes the feature usable with any type of data

Algorithm 8 Parallel MST with Distance Array and Reductions.

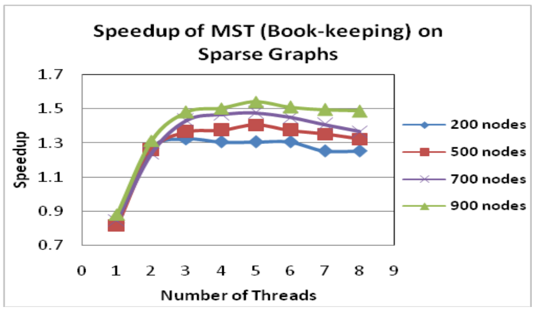
```
1:  Select a vertex v and let
    V(T)={v} and E(T)= {}
2:  while(true)
3:    PI = getIterator( incident edges of v)
4:    while(PI.hasNext){
5:       edge i = PI.next
6:       update d with i if i has weight less
         than what is stored in d
7:    }
8:    PI = getIterator( list of all nodes)
9:    minimumEdge = {}
10:   Reducible globalMinEdge = {}
11:   while(PI.hasNext){
12:     node v = PI.next
13:     edge = d[v]
14:     if( weight of edge < minimumEdge){
15:       minimumEdge = edge
16:     }
17:   }
18:   FinalMinimumEdge = globalMinEdge.reduce
19:   w = vertex of FinalMinimumEdge which is
      not visited
20:   Add w to V(T) and FinalMinimumEdge to E(T)
21:   If(size of E(T)=n-1) {
22:     exit the algorithm
23:   else
24:     v = w
25:   }
26:}
```

and any kind of reduction (Giacaman et al. 2008). In this approach, the PI reduction was customized in a way which returns an edge with the minimum weight.

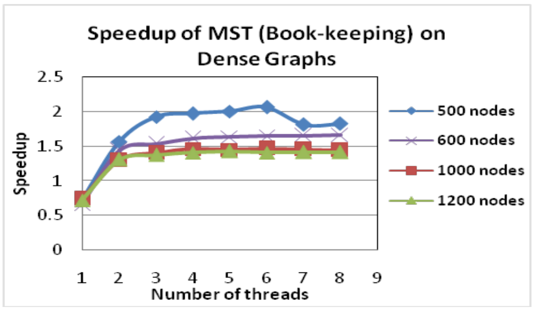
6.4 MST Performance Results

The performance of the two parallel MST approaches which were discussed in sections 6.3.1 and 6.3.2 was evaluated and compared with a large set of experiments on an 8-core machine. The experiments were run on different graphs with number of nodes and densities.

Figure 13 shows the speedup of the parallel MST with the book-keeping approach discussed in section 6.3.1 when tested on sparse and dense graphs.



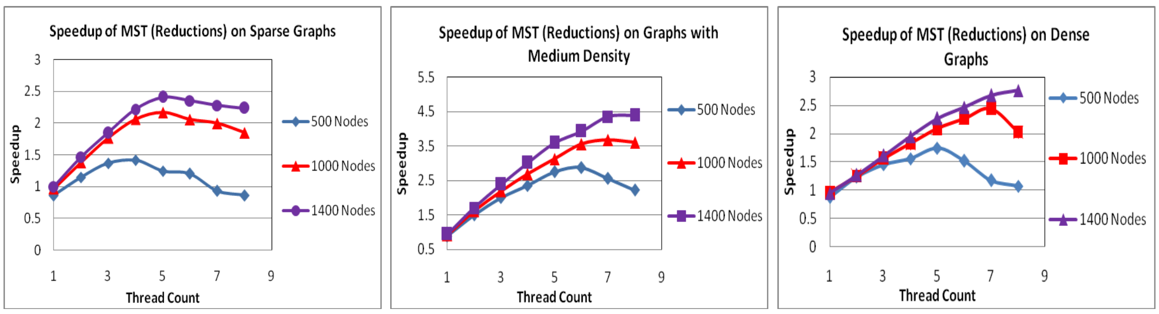
(a)



(b)

Figure 13: Speedup of MST with Book-keeping.

Figure 13a shows that the performance of the par-



(a) (b) (c)

Figure 14: Speedup of MST with Distance Array and Reductions.

allel MST gets better as the number of nodes in the sparse graph increases. However, since the graph is sparse, the produced speedup is not that significant (i.e. around 1.6 max). When the approach is tested on dense graphs as shown in Figure 13b , a slightly better performance is produced (i.e. reaches up to around 2.1). However, Figure 13b shows that as the number of nodes increases, the speedup gets worse when the approach is tested on dense graphs. This is the main limitation of this approach since running the sequential algorithm which connects the sub-trees produced in parallel slows down the run-time of the algorithm and produces bad performance on dense graphs as the number of nodes increases. This is because in dense graphs, the number of edges is equal to V^2, where V is the number of nodes in the graph. More edges result in more collisions and hence, more sub-trees to be merged sequentially.

Figure 14 shows the speedup results of the parallel MST approach with distance array and reductions discussed in section 6.3.2 when tested on graphs with low density (i.e. Sparse), medium density and high density.

The results in Figure 14 shows that the second parallel MST approach performs significantly better on dense graphs than the first approach illustrated in Figure 13. The approach produces the best performance when was tested on graphs with medium densities as the speedup reaches up to around 4.5.

7 Conclusions

Desktop applications must be parallelised to benefit from the introduction of multi-core processors. However, parallelising applications is considered to be a significantly challenging task. The PI is an OO tool which automates the process of parallelising loops in OO applications. Graphs are excellent use cases for the PI since they are naturally represented by objects. To maintain high productivity, readability and maintainability of OO graph algorithms, the parallelisation should be done in an OO way. Using the PI to parallelise graph algorithms is powerful in terms of producing a readable and maintainable code which exhibits speedup. Three main algorithms were parallelised using the PI: BFS, DFS and MST. Some parallel approaches of those algorithms required some adaptations and improvements to be added to the PI. The improvements include integrating a book-keeping module with the PI to perform concurrent colouring when running the MST and extending the *hasNext* and *next* methods of the PI to return nodes on the fly in breadth-first or depth-first order. Experimental results show that the algorithms which were parallelised by the PI exhibit good speedup while keeping the readability and maintainability of an OO code.

References

Akeila, L. (2008), Parallel iterator, Technical report, ECE department, University of Auckland.

An, L., Xiang, Q. & Chavez, S. (2000.), A fast implementation of the minimum spanning tree method for phase unwrapping, *in* 'Med. Imaging'.

Buckley, F. & Lewinter, M. (2003), *A Friendly Introduction to Graph Theory*, Prentice Hall/Pearson Ed.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms*, McGraw-Hill.

Craig, I. (2001), *The Interpretation of Object-oriented Programming Languages*, Springer.

D.Bader & Cong, G. (2004.), Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs, *in* 'Parallel and Distributed Processing Symposium'.

Giacaman, N., Sinnen, O. & Akeila, L. (2008), Object-oriented parallelisation: Improved and extended parallel iterator, *in* '14th IEEE International Conference on Parallel and Distributed Systems'.

Grama, A., Gupta, A., Karypis, G. & Kumar, V. (2003), *Introduction to Parallel Computing 2nd edition*, Addison-Wesley.

N.Giacaman & O.Sinnen (2008), Parallel iterator for parallelising object oriented applications, *in* '7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'08)'.

Rajasekaran, S. & Reif, J. (2007), *Handbook of Parallel Computing, Models, Algorithms and Applications*, CHAPMAN & HALL.

Reinders, J. (2007), *Intel Threading Building Blocks: Outfitting C++ for Multi-Core*, O'Reilly.

Reinefeld, A. & Schnecke, V. (1994.), Work-load balancing in highly parallel depth-first search, *in* 'Scalable High- Performance Computing Conference'.

Silvela, J. & Portillo, J. (2001), Breadth-first search and its application to image processing problems, *in* 'Image Processing', Vol. 10, pp. 1194–1199.

S.Meguerdichian, Koushanfar, F., Potkonjak, M. & Srivastava, M. (2001), Coverage problems in wireless ad-hoc sensor networks, *in* 'Proc. INFOCOM'01'.

Sun, J. (n.d.), 'Java concurrent queue, retrieved from http://java.sun.com/javase/6/docs/api/'.

Yooy, A., Chowx, E., Hendersony, K., McLendonz, W., Hendrickson, B. & Catalyurek, U. (2005), A scalable distributed parallel breadth-first search algorithm on bluegene/l.

Zhang, Y. & Hansen, E. A. (2006), Parallel breadth-first heuristic search on a shared-memory architecture, *in* 'Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications'.