Parallel Data Structures

ADVANCED SEMINAR

submitted by
Christoph Allig

NEUROSCIENTIFIC SYSTEM THEORY
Technische Universität München

Prof. Dr Jörg Conradt

Supervisor: Dipl.-Inf. Nicolai Waniek
Final Submission: 07.07.2015

TECHNISCHE UNIVERSITÄT MÜNCHEN
NEUROWISSENSCHAFTLICHE SYSTEMTHEORIE
PROF. DR. JÖRG CONRADT

2015-04-08

A D V A N C E D S E M I N A R

Parallel Data Structures

Problem description:

Literature on parallel algorithms and data structures is vast. In fact, the literature has grown with the advent of GPGPU computing just a few years ago. However, most of the literature targets SIMD (single instruction, multiple data) hardware, which means that data structures are still sequential at their heart.
In contrast to this, natural systems, and more recently new hardware, operates in a fashion which is asynchronous, solving MIMD or MISD (multiple instruction multiple data, multiple instruction single data) problems. As a consequence, we are interested in finding existing data structures that operate in an asynchronous fashion, and their benefits or drawbacks when compared to other data structures.

The student shall research the state of the art of data structures for MIMD and MISD problems. Hence, the student will have to

- Identify key "ingredients" for parallel data structures.
- Give an overview of the most relevant of these and reference the related literature.
- Point out example problems for which these algorithms are highly suited.

As a plus, the student may implement one or more of the data structures to show examples, or think about examples how to employ such data structures given special hardware, e.g. SpiNNaker.
This task requires

- a solid interest in parallel computing and
- some knowledge of theoretical computer science (e.g. O-Notation).

Supervisor: Nicolai Waniek

(J. Conradt)
Professor

Abstract

This thesis elaborates on the challenges of designing, benefits and drawbacks of parallel data structures. After a short overview about computer architecture the most relevant parallel data structures are surveyed. The main focus lies on techniques like recursive star-tree, recurrence equations, MIMD algorithm and combination of synchronous and asynchronous model that structure any problem that can be handled in the "divide and conquer" manner. These approaches achieve high performance and scalability. Furthermore we consider and evaluate the performance of several implementations of the parallel data structures stack and queue. The best performance is obtained by nonblocking algorithms. Finally we regard a programming model based on nondeterminism and emergence imitating natural systems. These systems solve complex problems and achieve robust behavior, even each individual is following its own rule. For all the abovementioned techniques are given some suitable applications.

I would like to thank Nicolai Waniek for having supported me all along this seminar. I am very grateful for his precious help to guide my research.

Contents

Chapter 1

Introduction

Due to the proliferation of multi-core processors in the last decade, new parallel data structures for parallel algorithms has been investigated. Parallel data structures are required for storing and organizing data accessed by multiple threads. Designing parallel data structures is more difficult than their sequential counterparts because threads executing concurrently may interleave their steps in many ways. To avoid arbitrary outcome concurrency has to be considered in designing parallel data structures. As a result of concurrency it is difficult to verify the correctness of an implementation. Thus verification techniques like theorem prover or model checking can be helpful. Designing parallel data structure also provides challenges with respect to performance and scalability. The layout of processors, memory, data in memory and the communication load all influence performance. The performance speedup S of a parallel algorithm in comparison to its sequential one is bounded by Amdahl's law [Amd67]:

$$S = \frac{1}{r_s + \frac{1-r_s}{P}} \tag{1.1}$$

where r_s denotes the work which can not be done in parallel and P the number of processors. Scalability is the ability of an algorithm to handle a growing amount of work in a capable manner [Bon00]. The $O(N)$ notation is used to classify a algorithm according to its scalability. Due to the fact that nowadays increasing capability of computers is achieved by using multi-core processors, the use of parallel algorithms and data structures is indispensable.

This paper begins with analyzing the most common computer architectures. After that the paper surveys the most relevant parallel data structures and its key components. Thereby their advantages and disadvantages are analyzed. Furthermore algorithms are introduced using parallel data structures for solving some example problems. Particularly, it is deferred to casting sequential data structures to concurrent ones, structures supporting to partition a complex problem into subproblems, synchronizing techniques, as well as a concept based on nondeterminism and emergence.

Chapter 2

Computer Architecture

According to Flynn's Taxonomy computer architecture as "the structure of a computer that a machine language programmer must understand to write a correct program for a machine"[FK+97] can be classified into four main categories:

- SISD (Single Instruction, Single Data)
- SIMD (Single Instruction, Multiple Data)
- MISD (Multiple Instruction, Single Data)
- MIMD (Multiple Instruction, Multiple Data)

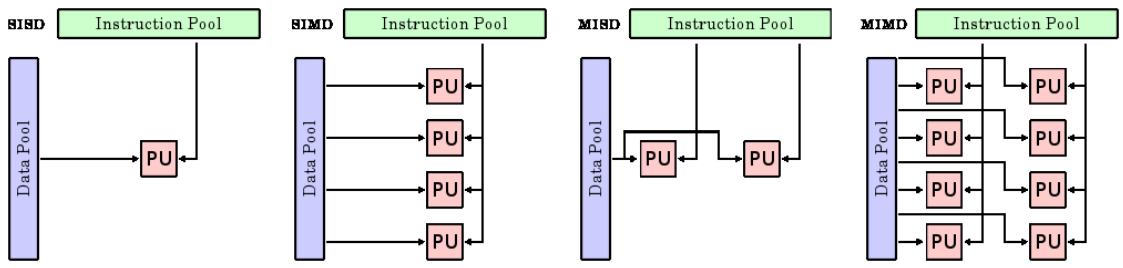


Figure 2.1: The four proposed computer architectures by Flynn's Taxonomy. (a) SISD, (b) SIMD, (c) MISD, (d) MIMD[FT1]

SISD is a sequential computer which exploits no parallelism neither in instruction nor in data streams. SIMD is a computer which exploits multiple data streams against a single instruction stream. For example, a GPU. The benefits of SIMD computer architecture are:

- simplicity of concept and programming
- regularity of structure

- easy scalability of size and performance
- straightforward applicability in a number of fields which demands parallelism to achieve necessary performance

In this work the focus will be on MISD and MIMD. MIMD architecture consists of a number of processors that function asynchronously and independently. According to this, different processors may be executing different instructions on different pieces of data. There are two types of MIMD architecture based on how MIMD processors access memory:

- Shared Memory MIMD architecture: Any processor can directly access any memory module via an interconnection network as observe on Fig. 2.2 (a).
- Distributed Memory MIMD architectures: Each pair of processor and memory are building a processing element (PE). The PE's are connected with each other via an interconnection network as observe on Fig. 2.2 (b).[KK13]

MIMD architecture is more flexible than SIMD or MISD architecture, but it's more difficult to create the complex algorithms that make these computers work. MISD

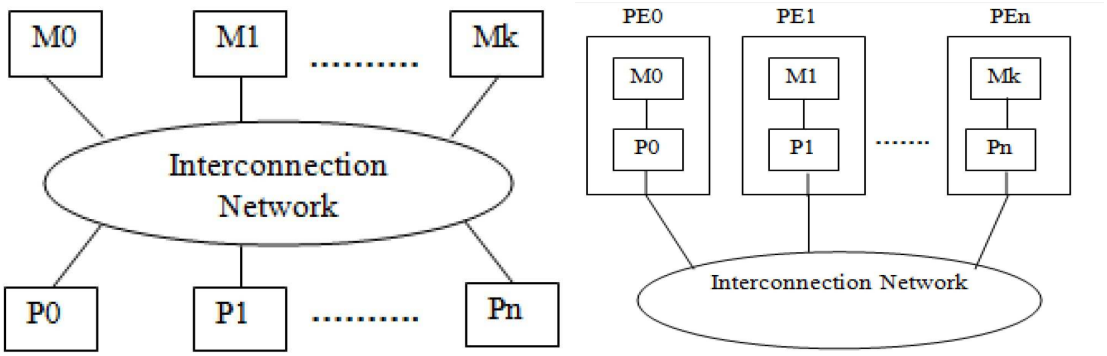


Figure 2.2: (a) Shared Memory MIMD Architecture (b) Distributed Memory MIMD Architecture[FK+97]

architecture consists of multiple processors. Each processor uses a different algorithm but uses the same shared input data. MISD computers can analyze the same set of data using several different operations at the same time. There aren't many actual examples of MISD computers, partly because the problems an MISD computer can calculate are uncommon and specialized.[PPA]

Chapter 3

Parallel Data Structure

3.1 Recursive Star-Tree

Berkman and Vishkin [BV93] introduced the recursive star-tree data structure which allows for fast parallel computations $O(\alpha(n))$ ($\alpha(n)$ is the inverse of the Ackermann function) using an optimal number of processors on a concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). Considering the assumption - a very small number ($O(log^{(i)}n)$) for any constant i of processors can write simultaneously each into different bits of the same word - these computations need only constant time. Several algorithms can be obtained by application of recursive star-trees:

1. lowest-common-ancestor (LCA) problem
2. level-ancestor problem
3. parenthesis matching problem
4. restricted-domain merging
5. almost fully-parallel reducibility

In the following the recursive star-tree data structure will be defined. A balanced tree $(BT)(m)$ with n leaves is a recursive tree in the sense that each of its nodes holds a tree of the form $BT(m-1)$. The number of levels in $BT(m)$ is $\star I_{m-1}(n)+1 = I_m(n)+1$. A node v at level $1 \leq l \leq \star I_{m-1}(n)+1$ has $\frac{I_{m-1}^{(l-1)}(n)}{I_{m-1}^{(l)}(n)}$ children, if $I_{m-1}^{(0)}(n) = n$. The total number of leaves in the subtree rooted at node v is $I_{m-1}^{(l-1)}(n)$. Fig. 3.1 displays the principle for a $BT(2)$.

To represent an application for the introduced data structure the LCA problem is analyzed. Given a sequence of n vertices $A = [a_1, ..., a_n]$ which is the Euler tour of the input tree and the level for each vertex v in the tree. The Euler tour, that starts and ends in the root of the tree, is obtained by replacing each edge (u, v) with two

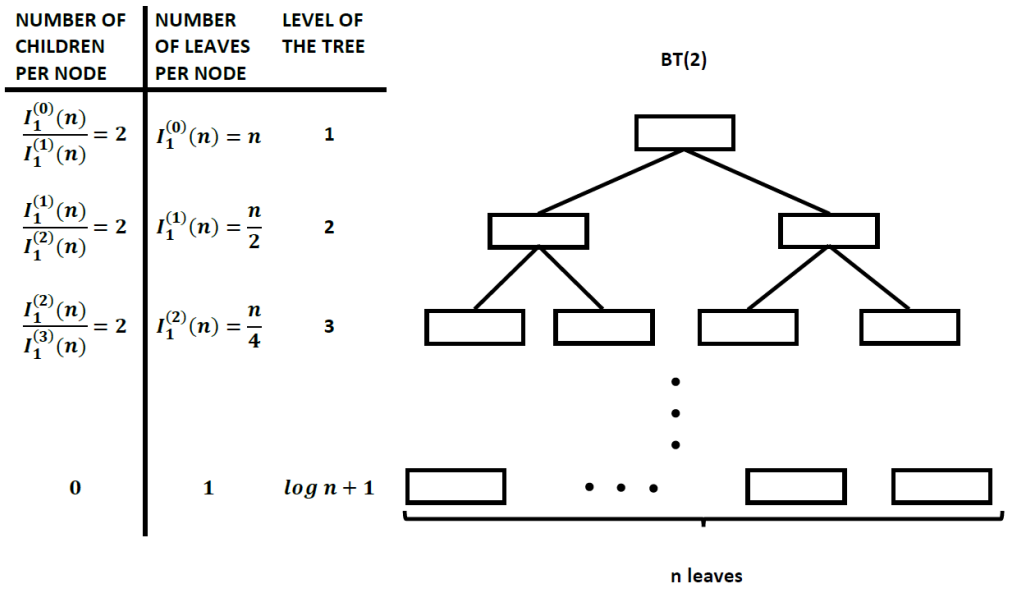


Figure 3.1: Balanced Tree $(BT)(2)$ is a complete binary tree with n leaves

edges (u, v) and (v, u) and computing the successor for each edge. Finally the Euler tour is stored in an array in a particular way to determine for each vertex in the tree its distance from the root. According to the corollaries:

1. vertex u is an ancestor of vertex v, if $l(u) \leq l(v) \leq r(u)$
2. vertices u and v are unrelated (neither u is an ancestor of v nor v is an ancestor of u), if $r(u) \leq l(v)$ or $r(v) \leq l(u)$
3. if u and v are unrelated vertices, the LCA of u and v is the vertex whose level is minimal over the interval $[r(u), l(v)]$ in A

the LCA problem can be solved by finding the following two indices in A:

- $l(v)$, the index of the leftmost appearance in A
- $r(v)$, the index of the rightmost appearance in A

For processing the corollary 3, a restricted-domain range-minima problem can be applied, because the difference between the level of each pair of successive vertices in the Euler tour is exactly one. The input for a general restricted-domain range-minima problem is the integer k and array $A = (a_1, a_2, ..., a_n)$ of integers, such that the difference between each $a_i, 1 \leq i < n$ and its successor a_{i+1} is at most k. The aim is to preprocess A so that any query $MIN[i, j], 1 \leq i < j \leq n$, requesting the minimal element over the interval $[a_i, ..., a_j]$, can be processed quickly. Below, there is a brief description for the preprocessing algorithm for $m = 2$.

1. Partition A into subarrays of $log^3 n$ elements each.

2. Take the minimum in each subarray to build array B of size $\frac{n}{\log^3 n}$.
3. Build a $BT(2)$, whose leaves are the elements of B. For each internal node v of $BT(2)$ there exists an array containing an entry for each leave of v in the subtree rooted at v.
4. Build two arrays of size n each, one contains all prefix-minima (between $l(v)$ and a leaf of v) and the other all suffix-minima (between a leaf of v and $r(v)$) with respect to A.

Generally the algorithm preprocess in cm time, for some constant c, using $nI_m^3(n) + \sqrt{k}nI_m(n)$ processors. Processing any range-minimum query can be done in cm operations.

Obviously, applying this data structure requires some additional effort to handle the coherencies in a suitable manner. On the other side is the proposed data structure applicable for a wide range of problems. It supports every problem that can be split into subproblems according to the "divide and conquer" manner. Algorithms using this data structure can be regarded as "optimal" due to $O(\alpha(n))$.

3.2 Stack and Queue

Most multiprocessors are programmed by time-slicing processors among processes to achieve acceptable response time and high utilization. Mutual exclusion locks degrades the performance significantly on time-slicing programmed systems due to the preemption of processes holding locks. In order to solve this problem two strategies have been proposed: preemption-safe locking and nonblocking algorithms. Preemption-safe locking techniques, e.g. [ELS88] or [MSLM91], allow the system either to avoid or to recover the adverse effect of the preemption of lock-holding processes. The nonblocking approach guarantees that at least one process of those trying to update the data structure concurrently will succeed in completing its operation within a bounded amount of time, regardless of the state of other processes. Generally, they employ atomic primitives, e.g. compare-and swap (CAS), load-linked (LL) and store-conditional (SC).

This chapter treats the parallel data structures Stack and Queue. In [MS98] and [MS07] can be found further information on parallel data structures in contrast to their sequential ones. There are treated: Heaps, Counters, Fetch-and-θ Structures, Pools, Linked Lists, Hash Tables, Search Trees and Priority Queues.

A concurrent queue - a First-In-First-Out (FIFO) data structure - provides enqueue and dequeue operations for addition and removal of entities. Fig. 3.2 [MS98] shows performance results for eight queue implementations on no-multiprogrammed system and on multiprogrammed systems with two and three processes per processor:

- usual single-lock algorithm using ordinary locks (single ordinary lock)
- usual single-lock algorithm using preemption-safe locks (single safe lock)
- two-lock algorithm using ordinary locks (two ordinary locks)
- two-lock algorithm using preemption-safe locks (two safe locks)
- nonblocking algorithm (MS nonblocking)
- PLJ nonblocking [PLJ94]
- Valois [Val95]
- Mellor-Crummey's (MC blocking) [MC87]

The execution time is normalized to that of preemption-safe single lock algorithm. The results show that as the level of multiprogramming increases, the performance of ordinary locks and Mellor-Crummey's blocking algorithm degrades significantly, while the performance of preemption-safe locks and nonblocking algorithms remains relatively unchanged. The two-lock algorithm allows more concurrency than the single-lock, but suffers more with multiprogramming when using ordinary locks because the chances are larger that a process will be preempted while holding a lock needed by other processes. With a multiprogramming level of 1 the two-lock algorithm outperforms a single-lock when more than four processors are active. The crossover point with levels of 2 and 3 occurs only for the preemption-safe locks at six and eight processors, respectively. The best performance is provided by the nonblocking algorithms, except of the Valois, which suffers from the complex overhead of the complex memory management. The nonblocking algorithms are privileged by no overhead of extra locks and invulnerable to interference from multiprogramming.

In the following the functionality of the nonblocking queue [MS98] is analyzed exemplary. Fig. 3.3 represents commented pseudocode for the structure and operations. The queue is constructed by a singly linked list consisting of Head and Tail pointers. Head points to a dummy node, which is the first node in the list. This allows more concurrency and simplifies the special cases associated with empty and single-item queues (a technique suggested by [Sit78]). Tail points to either the last or second to last node in the list. In order to avoid ABA problem CAS with modification counters is used. CAS compares the value of a shared memory location with the expected value. If both values are equal, the shared memory location is assigned a new value atomically. The ABA problem usually occurs when a thread reads a memory location twice and the read value is both A. Between the two reads another thread changes the value to B and restores the old value A. The first thread does not recognize the hidden modification, which might lead to incorrect behavior of the algorithm. The dequeue operation ensures that Tail does not point to the dequeued node or to any of its predecessors to allow dequeuing processes to free and then reuse dequeued nodes. Additionally, guarantying that values of pointers are unchanged is important for consistency.

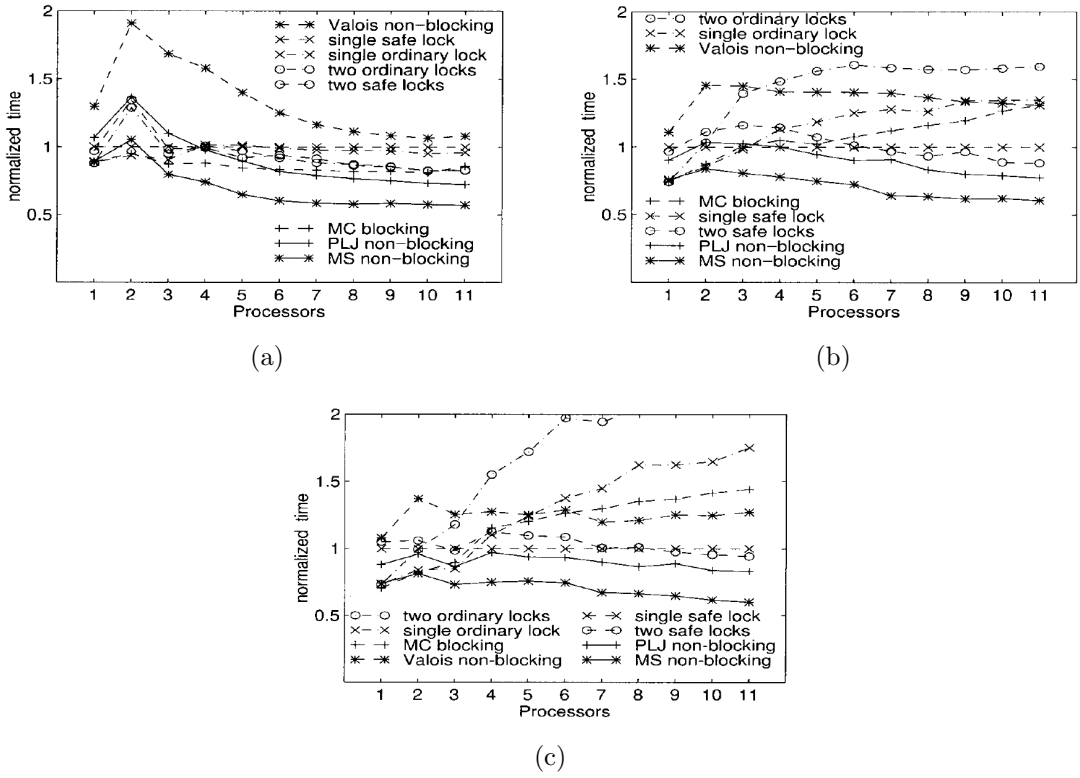


Figure 3.2: Normalized execution time for 1000000 enqueue/dequeue pairs on a multiprogrammed system, with multiprogramming levels of 1 (a), 2 (b), 3(c).[MS98]

A concurrent stack - a Last-In-First-Out (LIFO) data structure - provides push and pop operations for addition and removal of entities. Fig. 3.4 [MS98] shows performance results for four stack implementations:

- usual single-lock algorithm using ordinary locks (ordinary lock)
- usual single-lock algorithm using preemption-safe locks (preemption-safe lock)
- Treiber's nonblocking algorithm [Tre86] (Treiber non-blocking)
- optimized nonblocking algorithm based on Herlihy's general methodology [Her93] (Herlihy non-blocking)

As soon as the level of multiprogramming increases, the performance of ordinary locks degrades. The other three keep relatively constant. In all scenarios Treiber's algorithm achieves the best performance.

Treiber's as well as Herlihy's algorithm is implemented using a singly linked list including a pointer to the head of the list. Each process owns a copy of the Head. To successfully complete an operation the process has to use LL/SC to direct a shared pointer to the copied Head of the process. The shared pointer denotes the

```
structure pointer_t      {ptr: pointer to node_t, count: unsigned integer}
structure node_t         {value: data type, next: pointer_t}
structure queue_t        {Head: pointer_t, Tail: pointer_t}

INITIALIZE(Q: pointer to queue_t)
        node = new_node()                       # Allocate a free node
        node→next.ptr = NULL                    # Make it the only node in the linked list
        Q→Head.ptr = Q→Tail.ptr = node          # Both Head and Tail point to it

ENQUEUE(Q: pointer to queue_t, value: data type)
E1:     node = new_node()                       # Allocate a new node from the free list
E2:     node→value = value                      # Copy enqueued value into node
E3:     node→next.ptr = NULL                    # Set next pointer of node to NULL
E4:     loop                                    # Keep trying until Enqueue is done
E5:         tail = Q→Tail                       # Read Tail.ptr and Tail.count together
E6:         next = tail.ptr→next                # Read next ptr and count fields together
E7:         if tail == Q→Tail                   # Are tail and next consistent?
E8:             if next.ptr == NULL             # Was Tail pointing to the last node?
E9:                 if CAS(&tail.ptr→next, next, [node, next.count+1])  # Try to link node at the end of the linked list
E10:                    break                   # Enqueue is done. Exit loop
E11:                endif
E12:            else                            # Tail was not pointing to the last node
E13:                CAS(&Q→Tail, tail, [next.ptr, tail.count+1])  # Try to swing Tail to the next node
E14:            endif
E15:        endif
E16:    endloop
E17:    CAS(&Q→Tail, tail, [node, tail.count+1])  # Try to swing Tail to the inserted node

DEQUEUE(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:     loop                                    # Keep trying until Dequeue is done
D2:         head = Q→Head                       # Read Head
D3:         tail = Q→Tail                       # Read Tail
D4:         next = head.ptr→next                # Read Head.ptr→next
D5:         if head == Q→Head                   # Are head, tail, and next consistent?
D6:             if head.ptr == tail.ptr         # Is queue empty or Tail falling behind?
D7:                 if next.ptr == NULL         # Is queue empty?
D8:                     return FALSE            # Queue is empty, couldn't dequeue
D9:                 endif
D10:            CAS(&Q→Tail, tail, [next.ptr, tail.count+1])  # Tail is falling behind. Try to advance it
D11:            else                            # No need to deal with Tail
                    # Read value before CAS, otherwise another dequeue might free the next node
D12:                *pvalue = next.ptr→value
D13:                if CAS(&Q→Head, head, [next.ptr, head.count+1])  # Try to swing Head to the next node
D14:                    break                   # Dequeue is done. Exit loop
D15:                endif
D16:            endif
D17:        endif
D18:    endloop
D19:    free(head.ptr)                          # It is safe now to free the old dummy node
D20:    return TRUE                             # Queue was not empty, dequeue succeeded
```

Figure 3.3: Structure and operation of a nonblocking concurrent queue [MS98]

latest version of the stack. LL/SC, proposed by [JHB87], is used to atomically read, modify and write a shared memory location. LL returns the current value stored at the shared memory location. SC checks if any update have occurred to this shared memory location. If not then the shared memory location is updated successfully.

According to [HSY04] any stack data structure can be made more scalable using the elimination technique [ST97]. To apply this technique means that if a pop operation can find a concurrent push operation, then the pop operation can take the push operation's value and both operations can return immediately. So the pop operation eliminate the push operation, because eliminating both operations has the same effect on the state of the stack as if the push operation was followed immediately by the pop operation. The number of eliminations will grow with concurrency.

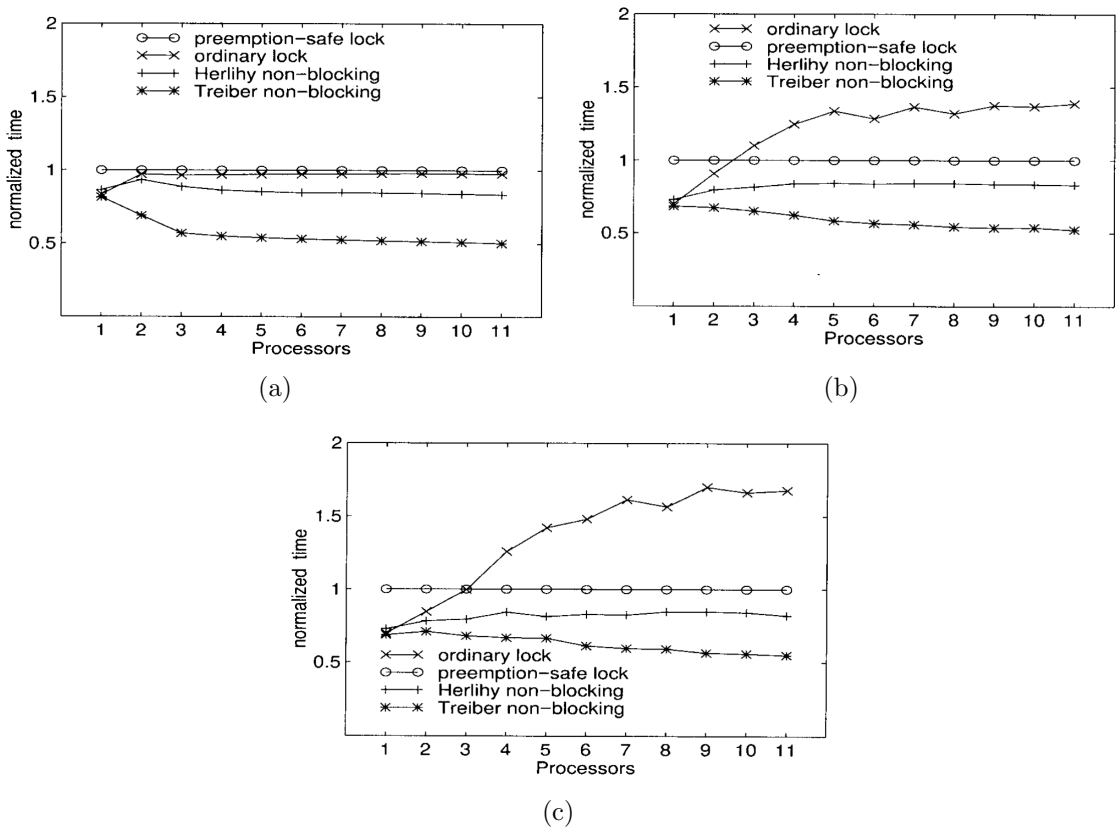


Figure 3.4: Normalized execution time for 1000000 push/pop pairs on a multiprogrammed system, with multiprogramming levels of 1 (a), 2 (b), 3(c).[MS98]

Chapter 4

Parallel Algorithm

4.1 MIMD Algorithm

[Wei87] proposed MIMD algorithms for solving the Schwarz Alternating Procedure and the Traveling Salesman Problem. The Schwarz Alternating Procedure solves a partial differential equation on a domain that is the union of two overlapping subdomains. In what follows the Traveling Salesman Problem shall be dealt. Given n cities and the euclidean distance between each pair, find the shortest cycle (closed tour) that contains each city exactly ones. [Qui83] proposed MIMD algorithms for the exact solution and approximation. As algorithms for the exact solution require a number of processors exponential in n, they are not relevant to practical use. Karp's partitioning method finds a closed tour through every city, but generally not the shortest:

1. halve the set of cities in x- or y-directions, depending on the larger extension. Both subsets have one common point
2. construct a closed subtour in each of the two subsets
3. combine the subtours by deleting two of the four edges incident to the bisection point and adding one new edge

Fig. 4.1 represent the procedure. Obviously the subproblems (step 2) can be computed independently. Only the straightforward step 3 needs some communication between the processes.

Considering the abovementioned instance, the general proceeding implementing a MIMD algorithm can be characterized as follows. The implementation consists of two steps. Depending on the logical relations, the first step divides the algorithm into parallel processes. The aim is to achieve the most possible parallelism with balanced computation and communication. The second step assigns the available processors to the parallel tasks. Obviously this method requires huge effort. Firstly a suitable parallelization has to be found for a particular problem. Secondly many details of

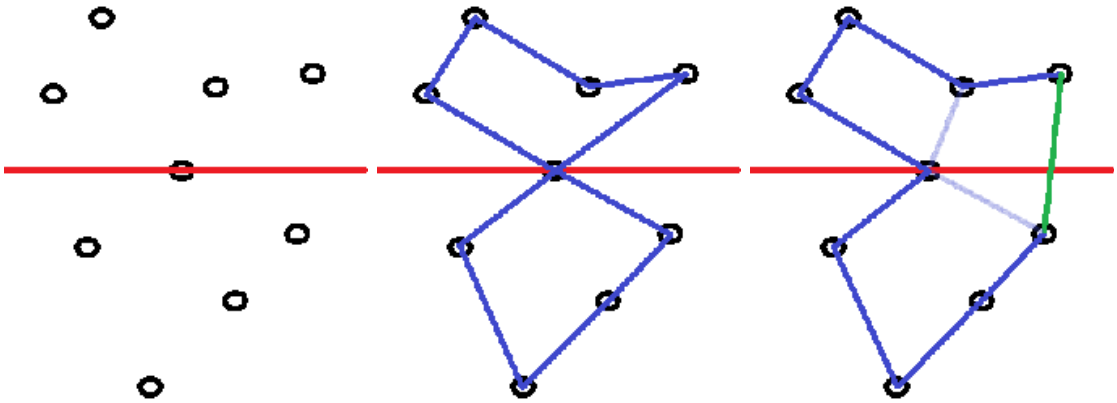


Figure 4.1: Karp's partitioning algorithm (a) 1. step (b) 2. step (c) 3. step

the computer system has to be considered to achieve a satisfactory performance.

4.2 Combination of Synchronous and Asynchronous Model

[SJ89] combines the benefits of a synchronous and asynchronous programming model. In SIMD architecture all threads execute the same instruction at the same time. So that at any given time each thread either executes the current instruction or sits idle for that instruction. On the other side SIMD programs are deterministic and avoid synchronization overhead. In a MIMD program each thread may be executing a different instruction simultaneously. The main drawback of MIMD is organizing communication among the threads. There are mutual exclusion (scarce resources can not be shared simultaneously) and producer-consumer problems (one thread needs the result of another). Typically that problems are not solved with data structure. Often used techniques for organizing the communication are semaphores, monitors, message-passing and pipes. The model of [SJ89] which uses data structures, additional compiler analysis, or run-time safety checks is proposed in the following. Program execution begins with a single thread performing arbitrary computations. At any time it may:

- quit itself
- create a set of sibling threads that are children of the creating thread
- execute an operation on shared memory

When a thread P creates a set of sibling threads then the execution of P is submitted and all the children begin execution asynchronously and in parallel. As soon as all children have quit themselves, P resumes its execution. All the threads that exists at any point in time are organized as a tree. Only the threads at the leaves are active. It is not assumed that all the active threads are equally complex. Operations on shared memory has to behave as serialized, because the interleaving of operations on shared memory by parallel is unpredictable. In order to guarantee that the unpredictability does not affect the externally visible behavior of a program some restrictions are necessary.

The two following techniques can be applied without injuring the previous defined model. For the purpose of a parent thread P executing in parallel with its children $p_1, p_2, ..., p_n$ an additional child thread p_0 has to be created to. p_0 continues the computation of the parents thread P. The other technique purposes suspending the children $p_1, p_2, ..., p_n$ of a thread P, executing P for a time, and then resuming $p_1, p_2, ..., p_n$. When $p_1, p_2, ..., p_n$ quit themselves, only the control threads are destroyed, but the corresponding state information of the children remain. Thereafter P can create new children $p'_1, p'_2, ..., p'_n$ continuing the computation of $p_1, p_2, ..., p_n$. In the proposed programming model the only allowed process synchronization is termination. This means that the parent thread knows that all of its children have been terminated, once the parent thread resumes operation.

In this model two operations E_1 and E_2 must not be causally related if there are two sibling threads p_1 that is responsible for E_1 and p_2 that is responsible for E_2. E operates on memory state M, changes the memory state to M' and returns value V. Two operations E_1 and E_2 are commute with respect to a memory state M if the order in which they are performed does not matter. Consequently $M''_a = M''_b$, $V_{1a} = V_{1b}$ and $V_{2a} = V_{2b}$ is required.

$$E(M) \rightarrow M' \Rightarrow V$$

$$\begin{aligned} &E_1(M) \rightarrow M'_a \Rightarrow V_{1a} \\ &E_2(M'_a) \rightarrow M''_a \Rightarrow V_{2a} \\ &E_2(M) \rightarrow M'_b \Rightarrow V_{2b} \\ &E_1(M'_b) \rightarrow M''_b \Rightarrow V_{1b} \end{aligned} \tag{4.1}$$

So the minimal restriction is that for any possible serialization order for the operations any two consecutive operations that are not causally related must commute with respect to the memory state. Simplifying this restriction results in that any two operations must be either causally related or commute. This restriction can be checked at compile time (static checking) or run time (dynamic checking).

Some state information that summarizes the history of operations is added to every cell. Cells are disjoint regions of the shared memory. The size of the history information is proportional to the maximum depth in the process tree of any thread that has been operating on the cell. The state information is an ordered tuple of pairs where every p_j is either a thread or the special marker *, and every e_j is either an operation class or the special marker *. p_j whose depth in the tree is j is responsible for the operation e_j on the cell. * indicates that more than one thread or operation class has been involved. Every thread also has linked with a set of cell, for which it is responsible for. When a thread is quit, all the cells in its responsibility set are added to its parent's responsibility set. As soon as a thread q at depth k resumes, all cells in the responsibility set for q must be pruned. In that way the history tuple for each cell is pruned to only the first k pairs.

The program will be aborted by the safety check if the program contains any two operations on the same cell that are not causally related and are not of the same operation class.

Even if this approach can support a large class of interesting problems, the exact range of applications is an open question. The technique can be applied for remaining a greater efficiency than a SIMD approach and greater safety than a MIMD approach. This is achieved by allowing asynchronous threads of control, but restricting shared-memory access.

4.3 Recurrence Equations

[KS73] proposed the technique recursive doubling for solving an mth-order recurrence problem of the form:

$$x_i = f_i(x_{i-1}, ..., x_{i-m}), \qquad i = m+1, ..., N \tag{4.2}$$

where x_i is a function of the previous m $x's$ $(x_{i-1}, ..., x_{i-m})$. The technique splits the computation into two equally complex subterms. Both subterms require the same number of instructions, each less than the original function, and types. Processing the two subterms can be performed in two separate processors. Continuous splitting distributes the computation over more processors. This technique yields in a computational time proportional to $\log_2 N$ distributed on N processors, whereas a serial algorithm computation time is proportional to N. P identical processors, each occupying its own memory, can communicate with every other. All processors obey the same instruction simultaneously, but any processor may be blocked from performing some instructions. Compared to already developed parallel algorithms for specific problems, like polynomial evaluation [MP73], recursive doubling benefits from its generality. It can be applied for any recurrence problem in the following form:

$$\begin{aligned} &x_1 = b_1 \\ &x_i = f_i(x_{i-1}) = f(b_i, g(a_i, x_{i-1})) \qquad 2 \le i \le N \end{aligned} \tag{4.3}$$

that satisfies certain simple restrictions:

1. f is associative: $f(x, f(y, z)) = f(f(x, y), z)$
2. g distributes over f: $g(x, f(y, z)) = f(g(x, y), g(x, z))$
3. g is semiassociative: there exists some function h such that $g(x, g(y, z)) = g(h(x, y), z)$

The general algorithm is given below. The vectors A and B consist of N elements. Processor (i) stores in its memory the ith component of each vector ($A(i)$ and $B(i)$). $A(i)$ and $B(i)$ may be scalars, matrices, lists, etc., depending on the problem. $A^{(k)}(i)$ and $B^{(k)}(i)$ represent respectively $A(i)$ and $B(i)$ after the kth step of the following algorithm:

- Initialization Step $(k = 0)$:

$$\begin{aligned} &B^{(0)}(i) = b_i \text{ for } 1 \le i \le N \\ &A^{(0)}(i) = a_i \text{ for } 1 < i \le N \\ &A(1) \text{ is never referenced and may be initialized arbitrarily} \end{aligned} \tag{4.4}$$

- Recursion Steps ($k = 1, 2, ..., \log_2 N$):

$$\begin{aligned} B^{(k)}(i) &= f(B^{(k-1)}(i), g(A^{(k-1)}(i), B^{(k-1)}(i - 2^{k-1}))) \quad \text{for} \quad 2^{k-1} < i \leq N \\ A^{(k)}(i) &= h(A^{(k-1)}(i), A^{(k-1)}(i - 2^{(k-1)})) \qquad\qquad\qquad\quad \text{for} \quad 2^{k-1} + 1 < i \leq N \end{aligned} \tag{4.5}$$

After $\log_2 N$th step $B(i)$ contains x_i for $1 \leq i \leq N$. This algorithm is merely able to process first-order recurrence equations. So as to be able to process also a mth-order recurrence equation, it has to be converted into a first-order equation using a slightly more complicated data structure. Appropriate applications comprise linear recurrence equations, polynomial evaluation, several nonlinear problems, the determination of the maximum or minimum of N numbers and the solution of tridiagonal linear equations.
In order to demonstrate the approach, consider the following first-order recurrence problem:

$$\begin{aligned} &\text{Given } x_1 = b_1, \text{ find } x_2, ..., x_N, \text{ where} \\ &\qquad\qquad x_i = a_i x_{i-1} + b_i \end{aligned} \tag{4.6}$$

We define the function $\hat{Q}(m, n)$, $n \leq m$:

$$\hat{Q}(m, n) = \sum_{j=n}^{m} \left(\prod_{r=j+1}^{m} a_r \right) b_j \tag{4.7}$$

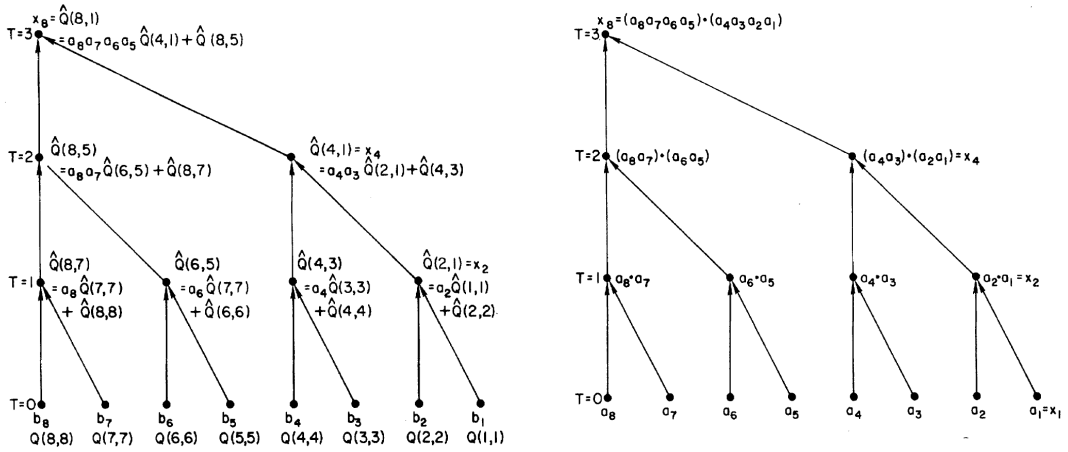
Now we write Eq. 4.6 by:

$$\hat{Q}(i, 1) = x_i \tag{4.8}$$

and apply recursive doubling:

$$\hat{Q}(2i, 1) = x_{2i} = \left(\prod_{r=i+1}^{2i} a_r \right) \hat{Q}(i, 1) + \hat{Q}(2i, i + 1) \tag{4.9}$$

Fig. 4.2 shows the computation graph of $\hat{Q}(8, 1)$.



(a) (b)

Figure 4.2: (a) Parallel computation of x_8 in the sequence $x_i = a_i x_{i-1} + b_i$. (b) Parallel computation of $x_8 = \prod_{j=1}^{8} a1$. [KS73]

4.4 Nondeterministic

[UA10] proposed a programming model, called Ly, that is based on nondeterminism and emergence. The basic concept of emergence is imitated from nature. Considering parallel systems, e.g. a flack of birds or an ant colony, a large number of individuals interact asynchronously. These systems solve complex problems and achieve coordinated and robust behavior, even each individual is following its own rules. Additionally there exists no global synchronization. In order to be able to implement such a programming model adverbs and ensembles with object-oriented inheritance are proposed. An ensemble consists of a flock of individuals. The members of an ensemble can be mutable objects or even other ensembles. The ensemble can be addressed as a whole. Performing an operation on it causes each member of it to perform that operation in parallel. In order to specify the following questions of execution strategy adverbs are applied. Which individuals of the flock should perform an operation? How shall the results, computed by each individual, be merged and returned? The adverb is appended to the list of arguments. Nevertheless an adverb is unnecessary for parallel operations on every individual, returning an ensemble of the results. Furthermore the ability performing an operation on a ensemble-as-a-whole is indispensable. Since often there is a decisive disparity.

In order to achieve a familiar programming model Ly is fundamentally constructed out of familiar syntax combined with object-inheritance. For this purpose the concept of an ensemble has to be integrated in object-based inheritance. It is proposed to allow every object to be an ensemble. Every object contains slots, a single link to its parent, and a one-to-many link to its members. A slot contains a name and (a

reference to) its content. A reference points to either an object or an ensemble. An object contains a parent reference and zero-or-more slots. An ensemble contains a parent reference and zero-or-more (references to its) members. Furthermore sending a message to an object causes a lookup in its ancestors if the object has no matching slot. The same applies to ensembles if its members do not understand the message. Otherwise sending a message to an ensemble will cause N parallel invocations, each per member.

The proposed approach has some undesirable effects. Firstly, considering scenarios of an empty ensemble is unsatisfying. For instance if a message is sent to an empty ensemble to do something, this can cause a different behavior than it is sent to a singleton ensemble. This is the case if the empty ensemble has an ancestor for that the method is defined differently. That implies braking the sake of consistency. Secondly, the question arises what to do when a message sent to an ensemble consisting of mutable members is understood by only some of its members. Thirdly, an operation consisting of an $if(true)\{\text{do something}\}$ condition processed by an ensemble returns N times $true$, dependent on the number of its members. As a result do something is performed N times of each member. Fourthly, the approach will not be applicable for many algorithms, because there is no ability for exchange between processes due to the lack of synchronization.

Chapter 5

Conclusion

This thesis surveyed existing parallel data structures that can be applied to achieve asynchronous algorithms. Generally, their design is more difficult due to concurrency, verifying correctness and scalability. However, once the challenges are hurdled parallel data structures constitute the basis to reach a maximum utilization of processors. Convenient parallel data structures obviate the need for synchronization techniques such as semaphores or monitors.

Most of the structures considered in this thesis are based on problems that can be solved recursively. The recursion is modeled by a tree and each of the eventuated subproblem is assigned to one processor. The processors solving the problems on one level can operate independently. Solely during merging the subproblems some interacting is needed. Though for all that the communication problem is solved by these data structures per se. After devoting huge effort to model the recursive problem in a convenient way, the algorithm results in high performance and scalability. The most general MIMD algorithm approach divides the problem into subproblems depending on their logical relations and assigns the available processors to the parallel tasks. By combination of synchronous and asynchronous model safety checks can be modeled to detect any two operations on a memory that are not causally related or commute. Both techniques are convenient to solve any problem that can be divided into subproblems. Due to their generality more effort is required to implement a proper parallel algorithm compared to the following. Recursive doubling can be applied for any recurrence equations that fulfill particular restrictions. Whereas the recursive star-tree is especially suited for search problems.

Furthermore this thesis considers the parallel data structures stack and queue in contrast to their sequential ones. To evade preemption of processes holding locks strategies such as preemption-safe locking and nonblocking algorithms employing atomic primitives are proposed and evaluated. The best performance is obtained by nonblocking algorithms.

A completely different approach based on nondeterminism and emergence imitates natural parallel systems, e.g. a flack of birds. These systems solve complex problems and achieve robust behavior, even each individual is following its own rule. A programming model is described using adverbs and ensembles in combination with object-oriented inheritance. However this proposed model is still immature.

List of Figures

Bibliography

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[Bon00] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[ELS88] Jan Edler, Jim Lipkis, and Edith Schonberg. *Process management for highly parallel UNIX systems*. Citeseer, 1988.

[FK+97] Terence FOUNTAIN, Peter Kacsuk, et al. *Advanced computer architectures: a design space approach*. Pearson Education India, 1997.

[FT1] Flynn's Taxonomy. `http://en.wikipedia.org/wiki/Flynn%27s_taxonomy`. Accessed : 2015-04-28.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[JHB87] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[KK13] Mandep Kaur and Rajdep Kaur. A Comparative Analysis of SIMD and MIMD Architectures. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(9):1151–1156, 2013.

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, 100(8):786–793, 1973.

[MC87] John M Mellor-Crummey. Concurrent queues: Practical fetch-and-θ algorithms. Technical report, Technical Report TR 229, Dept. of Computer Science, University of Rochester, 1987.

[MP73] Ian Munro and Michael Paterson. Optimal algorithms for parallel polynomial evaluation. *Journal of Computer and System Sciences*, 7(2):189–198, 1973.

[MS98] Maged M Michael and Michael L Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[MS07] Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[MSLM91] Brian D Marsh, Michael L Scott, Thomas J LeBlanc, and Evangelos P Markatos. First-class user-level threads. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 110–121. ACM, 1991.

[PLJ94] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *Computers, IEEE Transactions on*, 43(5):548–559, 1994.

[PPA] How parallel processing works. /urlhttp://computer.howstuffworks.com/parallel-processing1.htm. Accessed: 2015-04-29.

[Qui83] Michael Jay Quinn. Design and analysis of algorithms and data structures for the efficient solution of graph theoretic problems on MIMD computers. 1983.

[Sit78] RL Sites. *Operating systems and computer architecture*. 1978.

[SJ89] Guy L Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231. ACM, 1989.

[ST97] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30(6):645–670, 1997.

[Tre86] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[UA10] David Ungar and Sam S Adams. Harnessing emergence for many-core programming: early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 19–26. ACM, 2010.

[Val95] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[Wei87] P Weidner. MIMD algorithms and their implementation. In *WOPPLOT 86 Parallel Processing: Logic, Organization, and Technology*, pages 75–86. Springer, 1987.