Parallel Nondeterministic Programming as a Language Extension to C (Short Paper)

Lucas Kramer
krame505@umn.edu
University of Minnesota
Minneapolis, MN, USA

Eric Van Wyk
evw@umn.edu
University of Minnesota
Minneapolis, MN, USA

Abstract

This paper explores parallel nondeterministic programming as an extension to the C programming language; it provides constructs for specifying code containing ambiguous choice as introduced by McCarthy. A translator to plain C code was implemented as an extension to the ableC language specification. Translation involves a transformation to continuation passing style, providing lazy choice by storing continuation closures in a separate task buffer. This exploration considers various search evaluation approaches and their impact on correctness and performance. Multiple search drivers were implemented, including single-threaded depth-first search, a combined breadth- and depth-first approach, as well as two approaches to parallelism. Several benchmark applications were created using the extension, including n-Queens, SAT, and triangle peg solitaire. The simplest parallel search driver, using independent threads, showed the best performance in most cases, providing a significant speedup over the sequential versions. Adding task sharing between threads showed similar or slightly improved performance.

CCS Concepts • **Software and its engineering → Parallel programming languages**; **Extensible languages**.

Keywords nondeterministic programming, parallel programming, extensible languages

ACM Reference Format:
Lucas Kramer and Eric Van Wyk. 2019. Parallel Nondeterministic Programming as a Language Extension to C (Short Paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357765.3359524

1 Introduction

Nondeterministic programming is a semi-declarative programming paradigm, useful for expressing solutions to some types of search problems. Instead of needing to explicitly integrate all control logic into their code, nondeterministic programming allows programmers to express their logic in terms of new control constructs representing "ambiguous" choice and related operations. Issues related to search ordering and parallelism are managed by the language, allowing the programmer to experiment with various implementations much more easily. Such code should also be easier to write and debug, as programmers need only worry about the logic of their application, while relying on the semantics of the nondeterministic language to correctly implement and manage the search processes.

Efficiency is important in any programming paradigm, and is determined principally for nondeterministic programming by the approach taken to drive exploration of the search tree: should this be done in a depth-first order, breadth-first order, or something else? As expected, this is highly dependent on the particular application. However, in exploring new approaches it is also important to consider what search ordering semantics are provided; this essentially constitutes a "contract" between the language designer and the programmer, providing rules for what can and cannot be done; for example, can choice options be tried in a different order than they are written? Permitting such may allow for performance improvements, but may hinder the programmer in reasoning about correctness and termination. Parallel evaluation approaches may allow for the greatest speedups, but excessively strict constraints may hinder parallelism.

We explore nondeterministic programming by implementing the desired language constructs not as a new language but instead as a language extension to C, using the ableC extensible C language specification [Kaminski et al. 2017a][1]. This is useful for programmers using the system since many of the standard parts of any language (arithmetic expressions, I/O, etc.) simply come from the C host language. This also frees the language developers to focus their efforts on the new constructs being explored and not on re-implementing standard language features.

[1] ableC and the nondeterministic extension (with all the example programs) are both available at http://melt.cs.umn.edu and archived, respectively, at https://doi.org/10.13020/D6VQ25 and https://doi.org/10.13020/b42x-hm18.

```
1  search int range(int lower, int upper) {
2    choice for (int i = lower; i < upper; i++)
3      succeed i;
4  }
5  search int factor(int n) {
6    if (n % 2 == 0) {
7      choice {
8        succeed 2;
9        choose succeed factor(n / 2);
10     }
11   } else {
12     choose int a = range(ceil(sqrt(n)), n);
13     spawn;
14     float b = sqrt(a * a - n);
15     require b == floor(b);
16     choice {
17       succeed a - (int)b;
18       succeed a + (int)b;
19 } } }
20 ...
21 int result;
22 bool success = invoke(search_sequential_dfs,
23     &result, factor(n));
```

```
1  void range(closure<(int) -> void> k,
2             int lower, int upper) {
3    for (int i = lower; i < upper; i++)
4      k(i);
5  }
6  void factor(closure<(int)->void> k, int n){
7    if (n % 2 == 0) {
8      k(2);
9      factor(k, n / 2);
10   } else {
11     range(lambda (int a) -> void {
12         float b = sqrt(a * a - n);
13         if (b == floor(b)) {
14           k(a - (int)b);
15           k(a + (int)b);
16         }
17       }, ceil(sqrt(n)), n);
18 } }
19 ...
20 int result;
21 bool success;
22 factor(lambda (int res) -> void
23     { result = res; success = true; }, n)
```

Figure 1. (left) A simple nondeterministic program implementing Fermat's method of factorization to find one of the prime factors of an integer[2]. (right) A simplified translation of this program, ignoring the existence of spawn and tasks.

2 Design

Nondeterministic Extension to C: A simple example of a nondeterministic program is given on the left in Figure 1; it implements Fermat's method for factorization, based on the principle that any odd integer n may be expressed as the difference of two squares, giving

$$n = a^2 - b^2 = (a + b)(a - b) \tag{2.1}$$

We can use this to factor a number by nondeterministically trying a range of values for a, and computing b until it is found to be an integer.

The extension to C provides a number of new keywords: the `search` keyword marks the the definition of a *search function*, whose body is an embedded domain-specific language for nondeterministic programming. The syntax of a search function body mirrors ordinary C statements, with declarations, expressions, and control statements allowed.[3]. The most basic nondeterministic construct is the `choice { ... }` statement. When reached, execution "splits" for each statement within the choice (referred to as *options* of the choice) and continues along multiple virtual threads until success or failure.[4] On line 7 on the left execution either `succeed`s or tries `n / 2`. Often the number of options for a choice may vary dynamically, so we provide the

`choice for (init; cond; update) { body }`

statement, that spawns a new virtual thread for each iteration, see line 2. Note this differs from a normal for loop in that each iteration is independent from the others.

Execution of a search function may succeed (possibly many times) or fail; this is indicated by the `succeed val;` and `fail;` statements, used in place of `return`. Failure is silent and implicit: a search function may fail simply by ending without an explicit `succeed`. A common idiom is `if (cond) { ... } else fail;` failing if a condition is not met; thus an equivalent shorthand `require cond;` (line 15) is provided.

Another operation is to call a search function and nondeterministically select succeeding result value(s). This may be done with the `choose def = fn(args);` search statement (line 12), nondeterministically binding a variable to every succeeding value of the search function, each in a new virtual thread. Sometimes we only need one result; in these cases the `pick` statement is used. `pick` is identical to `choose`, except further exploration of choice options in the called search function will be canceled after the first success; this

[2] Only recording one result is currently supported; more can be obtained by invoking a wrapper search function that saves results to a buffer.
[3]Nondeterministic statements may not occur inside ordinary C iteration statements, for implementation reasons that will become apparent later - instead the program may need to be transformed to use recursion.

[4]By *virtual threads* we simply mean independent paths of execution, regardless of whether the implementation utilizes multithreaded parallelism.

is somewhat analogous to cut in Prolog. Since choosing or picking a value and immediately succeeding with it is a common pattern, the shorthand `choose succeed fn(args)` (line 9) or `pick succeed fn(args);` is provided, also allowing optimized translation.

By default, the virtual threads created by the options of a `choice` statement are fully evaluated in order, resulting in a sequential depth-first search. More flexibility is needed to allow for different evaluation approaches and parallelism; to accomplish this, a program may be divided into discrete tasks to be lazily executed in a potentially different order, or even in parallel by multiple real threads. A task is begun by the `spawn` statement, creating a new task for each virtual thread of execution reaching that point. Each task comprises all code executed until the next `spawn` or the end of the program.

Translation: The translation of a nondeterministic program into "plain" C code is required to execute the program. The translation of the nondeterministic program on the left of Figure 1 can be seen on the right of that figure. This is essentially a translation to continuation-passing style (CPS), a method of transforming general recursion into tail recursion. The transformation works by replacing the return value of a function with a function closure, known as a *continuation*, passed as a parameter to be called with the function's result. When calling another CPS function, any further work needed to compute a result from the result of the call must be done inside the continuation passed to the called function. A CPS program may perform a nondeterministic choice simply by calling the continuation more than once, as exhibited on the right in Figure 1 on lines 14 and 15. Alternatively, failure can be handled simply by not calling the continuation, as is done in the case of `require`. This can be seen in the use of `require` on line 15 (right), where the `choice` on lines 16—19 (left) is only evaluated when the condition `b == floor(b)` is true. Note that if additional statements existed after a `choice`, they could simply be wrapped in an additional continuation closure and called multiple times instead of calling the function continuation; however, this simple example does not have any such statements. The translation of the nondeterministic program on the left of Figure 1 can be seen on the right of that figure. Here we utilize an existing lambda-closure extension to ABLEC [Kaminski et al. 2017b], inspired by a simplified form of lambdas from C++11.

Allowing for Parallelism: CPS must be extended to deal with the spawning of tasks. A task is represented as a closure similar to a continuation, except with no result value parameter. When a task is spawned its closure is stored in a "schedule" task buffer, the details of which will be discussed later. An external search driver function (provided in a runtime library) will demand the execution of these tasks, possibly in some parallel manner. Every search function, task closure, and continuation is parameterized by a pointer to the current buffer, so that executing a task on a

```
typedef closure<(taskbuf_t *) -> void>
        task_t;
void factor(
    closure<(taskbuf_t *, int) -> void> k,
    taskbuf_t *tb, int n) {
  if (n % 2 == 0) {
    k(tb, 2));
    factor(k, tb, n / 2);
  } else {
    range(
      lambda (int a, taskbuf_t *tb)
             -> void {
        put_task(tb,
          lambda (taskbuf_t *tb) -> void {
            float b = sqrt(a * a - n);
            if (b == floor(b)) {
              k(tb, a - (int)b);
              k(tb, a + (int)b);  }
          });  },
      ceil(sqrt(n)), n);
} }
```

Figure 2. The transformation of the program in Figure 1 using task closures and an external buffer.

buffer may place additional tasks in that buffer. Note that the continuation closures require this parameter as well, since a search driver may use multiple buffers internally, and a continuation may be executed on a different buffer than the one through which it was created. This transformation is shown for the factorization example in Figure 2.

To initially call a search function from a regular one, the `invoke` expression is used (line 22 of Figure 1), parameterized by the search driver function to use (and any parameters it takes), a pointer to a location in which to place the result, the search function to call, and the parameters. The result of this expression is a boolean value indicating whether the search succeeded. This expression simply translates to a call of the search driver with an initial task closure wrapping a call of the invoked search function. The continuation given to this search function records the result and signals that any remaining search may be canceled.

With the proliferation of tasks closures containing partial computation results, automatic memory management is vital; this was implemented via reference counting, although other approaches (such as the mark-and-sweep method provided by the Boehm GC [Boehm 1993]) could also suffice.

Search Ordering and Driver Functions: An important design issue is the order in which the search tree is to be explored; a balance must be struck between allowing the programmer to reason about correctness and termination, and allowing the search driver flexibility to introduce parallelism.

In breadth-first search (BFS), all tasks at depth n are evaluated in order before the first task at depth $n+1$ is evaluated. This makes reasoning about termination easy, as breadth-first evaluation means it cannot get "stuck" in a non-terminating subtree to the left of a solution - if a solution exists it will eventually be found. However, assuming a constant branching factor of k, if the first success occurs at depth n we will need to evaluate between k^{n-1} and k^n tasks before success. Additionally, we must store the entire final row of k^{n-1} tasks in the buffer before any can be evaluated, resulting in exponential memory usage. This algorithm is straightforward to implement by using a FIFO queue for the task buffer, and parallelism can be added simply by having multiple threads operating on the same buffer in parallel. However, the large memory overhead precludes the implementation of this algorithm for non-trivial applications.

In pure depth-first search (DFS), every branch of the search tree is fully explored before the next is tried. As this approach explores the entire tree from left-to-right, we may explore anywhere between 0 and k^n choices; although assuming a balanced tree with s randomly distributed leaf solutions, we will explore on average k^n/s choices before finding one, again assuming a branching factor of k and solution depth of n. However, a large or infinite subtree may occur before the first solution and the search would become "stuck", unlike in breadth-first search. On the other hand, the memory usage is greatly improved, since at most we only need to store nk tasks, all the options for the immediate ancestors of the succeeding task. The implementation is similar to BFS, though the buffer must be slightly more complex, as utilizing a simple LIFO stack would reverse the order in which options at each level are considered. Instead, we can use a "stack of queues," with each stack "frame" storing a queue of the options of a choice. Simple DFS unfortunately cannot be parallelized, as at every step in the search the next task is usually a descendant of the latest evaluated task.

In modified depth-first search (MDFS), we initially perform BFS down to a fixed level in the tree, then perform a simultaneous DFS of each resulting leaf until one succeeds. It can easily be shown that any program terminating under DFS will also terminate under MDFS. Though the actual performance may vary, on average MDFS explores a similar number of nodes as DFS, but MDFS can tolerate some threads becoming stuck in large or infinite subtrees. MDFS can be implemented by first using the BFS algorithm for a given number of iterations, before allocating a new buffer for every task in the queue. We then alternately perform a step for each of these queues until one succeeds. This sequential approach can easily be parallelized, by assigning each task in the queue to a thread at some level of the initial BFS.

A slight inefficiency is that some threads may exhaust their buffer and finish before others are done, preventing parallelism for long-running threads. This can be remedied by a *task-sharing* approach, where threads with tasks regularly check for (and redistribute tasks to) threads that have exhausted their buffers. When the last thread notices that it is about to wait for a task, the search fails. Alternatively, in a *task-stealing* approach, a thread, upon exhausting its buffer, takes the initiative in selecting another thread and stealing a task from that thread's buffer.

Four search drivers were implemented: single-threaded DFS and MDFS, MDFS with independent threads, and MDFS with task sharing threads (a task stealing version is left as future work.) All parallelism was done explicitly using pthreads; a fixed number of threads were spawned, and a global buffer synchronized by pthread mutexes was used to initially distribute tasks. The number of threads to use and the depth to perform BFS initially in each thread were also made parameters to the search drivers.

In the task sharing driver, some method is needed to wake up threads that are waiting for a task, once one has been provided. This was done via pthread *condition variables*, a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied [Leroy 2003].

3 Benchmark Applications and Results

In addition to the factorization example, we have implemented 3 benchmark applications, 2 of which are NP-complete problems:[5] n-Queens completion, triangle peg solitaire, and Boolean satisfiability (SAT), all available at the previously mentioned URLs.

n-Queens Completion: n-Queens is a famous problem in which n queens must be placed on an $n \times n$ board such that none are threatening another. This is a common AI benchmark that is actually solvable by heuristic. n-Queens completion is a slight modification, where some queens are initially placed on the board, and we wish to determine whether there is a solution. This variant is NP-complete [Gent et al. 2017], and is more computationally interesting.

The implementation of n-Queens consists of a recursive solver that nondeterministically chooses and applies moves. The logic for finding a valid move is shown in Figure 3. Note the use of `pick` in choosing an empty row; any empty row is an equally valid choice, but if we fail while considering a particular empty row, choosing a different row cannot lead to a solution since every row must eventually contain a queen. This problem has a relatively high branching factor of n for an $n \times n$ board, but a very low amount of computation is

[5]NP-complete problems are generally good candidates for nondeterministic programming. All NP-complete problems have a polynomial-time verification algorithm, so we could easily construct a nondeterministic program by ambiguously choosing any possible solution recursively and requiring that it is correct (although in practice we can do better with ordering heuristics and non-leaf constraints.) Although performance will still obviously be exponential, we may still be interested in a general but efficient solution for small problem sizes.

```
search unsigned empty_row(state_t st) {
  choose unsigned row = range(0, st.size);
  require !row_taken(st, row);
  succeed row;
}
search move_t valid_move(state_t st) {
  pick unsigned row = empty_row(st);
  choose unsigned col = range(0, st.size);
  require !col_taken(st, col);
  require !diag_taken(st, row, col);
  succeed (move_t){row, col};
}
```

Figure 3. A portion of the implementation of n-Queens.

needed at each node in the search tree to perform a move and compute the next valid moves. Due to the size of the board, a backtracking state representation was utilized to avoid excessive memory allocation and copy overhead.

In addition to the nondeterministic implementation, a plain C sequential version performing DFS was created. By comparing the performance of this with the performance using the DFS sequential driver, the performance overhead of the nondeterministic extension can be measured.

Triangle Peg Solitaire: Triangle peg solitaire is a simple game where the goal is to remove all but one peg in a triangular board by repeatedly jumping a peg over another into an empty hole on the board and then removing the jumped peg. The board starts with one empty hole. This problem has an intermediate branching factor, since there are often more than 2 available moves, and an intermediate workload at each node in the search tree, to compute the valid moves.

The solver is implemented as a search function that chooses valid move, applies it to the current state, and recursively solves the new state[6]. When a state with depth equal to the number of pegs is reached, a buffer to hold the results is allocated and returned, and each level the taken move is recorded in the buffer before it is returned. The implementation is not shown due to its complexity, but it is similar to that for n-Queens Completion and contains several `choice`s for finding potential moves and choosing one.

Boolean Satisfiability: Boolean satisfiability, or SAT, is another famous NP-complete problem. Although intractable in general, with the aid of heuristics it is possible to solve via search for some reasonably-sized instances.

A SAT solver was implemented for formulas represented in conjunctive normal form, using the following algorithm:

[6]The board state can be represented by a 64-bit integer, treated as a bit vector indicating whether each position is occupied. This limits the solver to board sizes smaller than 11. However boards larger than 9 are nearly intractable via brute-force search and can be solved via induction from smaller boards [Bell 2008], and thus are not computationally interesting.

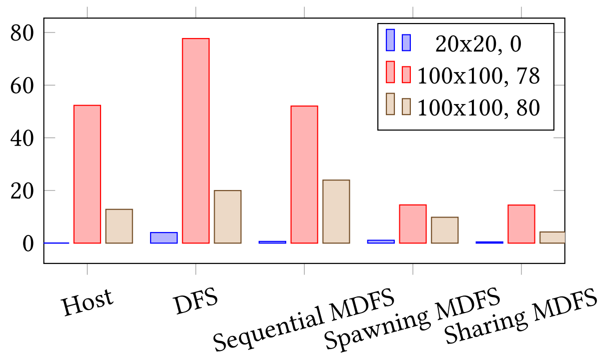

Figure 4. Measured runtimes in seconds for n-Queens with the "host" C implementation and various nondeterministic drivers. Legend: size x size, number of pre-filled locations. All but the 100x100,80 benchmark are solvable.

1. Simplify the formula to eliminate singular clauses (these are clauses with only one literal.)
2. Pick a variable to assign using the heuristic of having the same quality (negated or not negated) in the most clauses, and nondeterministically add it to the formula as either negated or not negated.
3. Recursively solve the resulting formula.

Due the overhead of the simplification phase, this problem has a high workload at every node in the search tree, and a small branching factor because there are only 2 assignments possible for each variable.

Performance Results and Discussion: The factorization example in Figure 1 was tried for a (relatively small) input of 21,641,161, the product of 2 primes. With all search drivers, the solution required about 3.6 seconds and 4GB of memory,[7] so larger tests were not attempted. This is because the initial "range" choice must generate a task for every number to be considered before any can be evaluated. Consequently, this extension is better suited to recursive search problems, where smaller choices are made throughout the search tree.

For n-Queens, 3 problems were evaluated: a 20x20 empty board, a 100x100 solvable board with 78 randomly-placed queens, and a 100x100 unsolvable board with 80 randomly-placed queens, arranged in this order in Figure 4, grouped by five different driver functions. Using the "spawning" and "sharing" MDFS drivers with initial depth 2, approximately 4x speed-ups were observed over the sequential MDFS driver.

For triangle peg solitaire, performance data was collected for boards of size 8 and 9, with the corner hole unoccupied.

[7] For all applications, performance data was collected on an Intel i7-8550U processor at 1.8 GHz. This processor features 8 logical cores (4 physical, 2x hyperthreading), 32KB L1 data and instruction caches, 256KB L2 cache, and 8.2MB L3 cache. For each data set, the runtime was measured for each of the drivers using a range of parameters, and averaged over 10 trials. All trials of parallel drivers were run with 8 threads. A range of initial depths were tried for the MDFS drivers and the fastest time was used.

With the parallel MDFS drivers, a speedup of approximately 3x was observed over sequential MDFS, using initial depth 7. The performance was virtually identical between spawning and sharing, because in both cases a success occurs before any threads run out of work, and thus no task sharing actually occurs. In fact, the sharing performance was marginally worse due to the additional overhead.

Similar speedups were obtained on the SAT benchmark. Two formulas from the SATLIB benchmark suite [Hoos 2000] were evaluated: a satisfiable formula containing 150 variables and 645 clauses (uf150-02) and an unsatisfiable formula containing 125 variables and 538 clauses (uuf125-01). For another unsatisfiable SAT formula a surprising 170x speedup was obtained for a satisfiable formula; more investigation is needed to fully understand this result.

For all applications the task-sharing MDFS driver showed the best or near-best performance, showing slightly more overhead than the spawning driver but beating it on exhaustive searches for problems lacking a solution. The optimal parameters tend to vary for the initial breadth-first search depth, although this could be addressed by heuristics or runtime auto-tuning approaches. For n-Queens, the plain-C implementation of sequential DFS was about 40% faster than the equivalent nondeterministic version using the sequential DFS driver, indicating a significant but tolerable amount of overhead. However the nondeterministic version still provided an effective performance speedup with parallelism.

4 Related Work

The concept of "ambiguous" or nondeterministic functions, and nondeterministic choice with regard to program flow, was first introduced by [McCarthy 1963]. In this case, nondeterministic choice is done at the expression level, via a new `amb` operator. Further inspiration on the language design came from Ableson et al. [1996] and AI lecture notes by Davis [2003]. Continuation-passing style is a long-standing technique in functional programming, and its use in implementing nondeterminism is also discussed in Ableson et al. [1996].

Parallel nondeterministic computing has widely been explored in logic programming, such as Parlog [Clark and Gregory 1986], which requires many of the same considerations with respect to search ordering. However, the automatic parallelization of an imperative nondeterministic language appears to be an original contribution by this paper.

Work sharing [Blumofe and Leiserson 2006] and work stealing [Blumofe and Leiserson 1999] are both existing approaches to ensuring even distribution of tasks when implementing parallelism. Work stealing has been used to implement schedulers for automatic parallelism frameworks, such as OpenMP [Olivier et al. 2012].

Previously language features and extensions have been proposed for providing task-based parallelism with work stealing, most notably Cilk [Frigo et al. 1998]. Spawn in the nondeterministic C extension is similar to spawn in Cilk, except that there is no corresponding *sync* operation; execution simply continues independently along each thread until failure or overall success.

ableC [Kaminski et al. 2017a] is an extensible specification of C using the Silver [Van Wyk et al. 2010] attribute grammar system. Silver provides guarantees of composability between independently developed language extensions by providing a modular determinism analysis [Schwerdfeger and Van Wyk 2009] for concrete syntax and a modular well-definedness analysis [Kaminski and Van Wyk 2012] for attributes on abstract syntax. The Silver-ableC [Kramer et al. 2019] extension to Silver was used to specify generated code with the concrete object-language syntax of ableC.

5 Conclusion and Future Work

This paper is a first step in the exploration of nondeterministic programming implemented in an imperative language as an extension to C. We introduce a number of language constructs for specifying these types of computations and compare multiple evaluation strategies. Significant performance improvements were found to be possible by leveraging parallelism. The overhead of the extension was found to be reasonably acceptable by comparison with a plain C implementation of an example application, although it is important to remember that goal of the extension is ease of development rather than absolute performance.

The collection of language constructs has been sufficient for the applications we have developed so far. However we continue to explore more intuitive and expressive abstractions. One area of interest is a more intuitive means to specify tasks than is done with `spawn` now. We are also considering abstractions that provide more fine grain control of the parallel evaluation than simply choosing a fixed driver function.

All trials were performed using 8 threads on a machine with 4 cores and 2x hyperthreading, so it would be interesting to collect scalability data on a machine with a larger number of cores. With a larger number of threads, the initial sequential phase of spawning MDFS would become more significant, potentially allowing the task sharing approach to win out in more situations with this increase in overhead.

Determining the best values for parameters such as the initial breadth-first search depth is open problem we anticipate investigating further. We expect auto-tuning techniques [Naono et al. 2010] to be particularly applicable.

Acknowledgments

This material is partially based upon work supported by the National Science Foundation (NSF) under Grant Nos. 1628929. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

References

Harold Ableson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2 edition, 1996. Description of nondeterministic computing in section 4.3.

George I. Bell. Solving triangular peg solitaire. *Journal of Integer Sequences*, 11, November 2008.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 56(5):720–748, September 1999.

Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, July 2006.

Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 197–206. ACM, 1993.

Keith Clark and Steve Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

Ernest Davis. Non-deterministic algorithms. cs.nyu.edu/courses/spring03/G22.2560-001/nondet.html, 2003. [Online; accessed May 3, 2018].

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 212–223, New York, NY, USA, 1998. ACM.

Ian P. Gent, Christopher Jefferson, and Peter Nightingale. Complexity of n-queens completion. *Journal of Artificial Intelligence Research*, 59:815 – 848, September 2017.

Holger H. Hoos. Satlib - benchmark problems. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html, August 2000. [Online; accessed May 3, 2018].

Ted Kaminski and Eric Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE)*, volume 7745 of *Lecture Notes in Computer Science*, pages 352–371. Springer, September 2012.

Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98:1–98:29, October 2017a. ISSN 2475-1421.

Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C — supplemental material. Technical Report 17-009, University of Minnesota, Department of Computer Science and Engineering, 2017b. Available at https://www.cs.umn.edu/research/technical_reports/view/17-009.

Lucas Kramer, Ted Kaminski, and Eric Van Wyk. Reflection in attribute grammars. In *Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE)*. ACM, 2019.

Xavier Leroy. *PTHREAD_COND(3) Linux User's Manual*, 2003.

John McCarthy. A basis for a mathematical theory of computation. *Computer programming and formal systems*, pages 33–70, 1963.

Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors. *Software Automatic Tuning*, 2010. Springer. ISBN 978-1-4419-6934-7. doi: 10.1007/978-1-4419-6935-4.

Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124, May 2012.

August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210, New York, NY, USA, June 2009. ACM.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.