4-2022

Parallel Resource Defined Fitness Sharing: A Study on Parallel Optimizations for Niching Algorithms

Blayne A. Rogers
Western Michigan University

PARALLEL RESOURCE DEFINED FITNESS SHARING: A STUDY ON PARALLEL OPTIMIZATIONS FOR NICHING ALGORITHMS

Blayne A. Rogers, M.S.

Western Michigan University, 2022

The exploitation of niches by genetic algorithms (GAs) is a computationally expensive, but effective, methodology for solving complex open problems and real-world applications. Niching, differentiated on the modality of sharing, casts problems in terms of the specific resources available. These concepts arise from the broader natural algorithms that encapsulate the ideas and theories used in artificial intelligence. In remediating the computational costs, a study on exploiting niche-defined parallel structures is performed in the contest of the resource-defined fitness sharing (RFS) algorithm.

Sharing is a natural algorithm paradigm that emulates the use of resources within an environment or population. Defining these resources presents two closely related techniques: resource and fitness sharing. Resource sharing seeks to apply the finite resources available, whereas fitness sharing assesses a population based on merit.

Resource and fitness sharing exhibit a duality within the sharing paradigm. Consequently, problems that have resource-defined niches are incompatible with fitness sharing. Conversely, resource sharing has great difficulty in managing the non-linear interactions among shared fitnesses. The RFS algorithm was developed to resolve these deficiencies and is the focus of this study for the parallel optimization of niching structures.

PARALLEL RESOURCE DEFINED FITNESS SHARING: A STUDY ON PARALLEL OPTIMIZATIONS FOR NICHING ALGORITHMS

by

Blayne A. Rogers

A thesis submitted to the Graduate College
in partial fulfilment of the requirements
for the degree of Master of Science
Computer Science
Western Michigan University
April 2022

Master's Thesis Committee:

Ajay Gupta, Ph.D., Chair
Elise de Doncker, Ph.D.
Jeffrey Horn, Ph.D.

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor, Dr. Gupta, of the Computer Science Department at Western Michigan University. He provided me with many opportunities to perform and expand upon my abilities to research. Also, his guidance was integral in flushing out important ideas within my research areas and how to explore them.

I would like to acknowledge Dr. de Doncker of the Computer Science Department at Western Michigan University as well. Her guidance on parallel computing and HPC topics were paramount to the completion of this thesis. Furthermore, her feedback and reviews were valuable in identifying weaknesses in my research.

Additionally, I would like to acknowledge Dr. Horn of the Math and Computer Science Department at Northern Michigan University. He presented me the opportunity to research the parallelization of the RFS algorithm. Moreover, he always found time to help clarify my understanding in the genetic algorithm domain, even into my master's degree.

Finally, I would like to express my deepest gratitude to my wife and children. Even with the many days and nights I was not available, they supported me throughout my education and also urged me to continue into the Ph.D. program at Western Michigan University.

Blayne A. Rogers

TABLE OF CONTENTS

Table of Contents—Continued

CHAPTER

Table of Contents—Continued

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis performs an incremental, in-depth analysis of the resource-defined fitness sharing (RFS) algorithm (Horn, 2002). This is classified as a parallel niche genetic algorithm p(NGA). There are two subcategories of genetic algorithms to address here.

First, parallel genetic algorithms are those that (A) can exploit parallel hardware architectures, (B) can exploit two or more genetic algorithms to find a solution, and (C) are designed to find multiple solutions. The other category the algorithm falls into is the NGAs. These genetic algorithms use some form of niching to cast problems into explicit or implicit resources.

There are two general techniques used to induce niching, which are resource sharing and fitness sharing. Resource sharing casts problems into explicit resources, whereas fitness sharing rewards individuals in a defined population based on how accurately they solve a problem. In other words, fitness sharing rewards individuals based on merit.

The RFS algorithm combines the techniques to resolve limitations inherent in both. Fitness sharing cannot manage resource-defined niches. In contrast, resource-sharing cannot resolve the non-linear dynamics imposed by fitness sharing. The RFS algorithm achieves the amalgamation of these techniques by casting problems into explicit resources, while maintaining simple linear equations for exploiting fitness within the population.

While the algorithm has shown significant empirical evidence to its capabilities, especially in the exact cover domain, it is limited by the natural weaknesses inherent in both sharing techniques. These include the insufficiency of local search and slow convergence in later generations (Gu et al., 2007).

Limitations due to local search are addressed by maximizing the available information of the problem domain. The added information is exploited to better refine both the algorithm's

injection point into the solution space and its ability to search the space to find an answer by the time niche equilibrium is achieved.

Consequently, the analysis performed in this thesis focuses on resolving the inherent limiting factors of spatial and temporal complexity imposed by the algorithm and problem domain. These factors can further be expanded to address issues involving serial application, memory, data structure, and computational inefficiencies, as well as the computational noise permeated by the interactions of species. In addressing these limitations, we seek to reduce the amount of time per generation and the number of generations required to achieve convergence.

While the complexity analysis is primarily performed on the RFS algorithm, the issues presented in both spatial and temporal complexity were found to be dominated by the problem domain. Indeed, the algorithm itself is generalized enough to absorb the complexities of the problem domain with little injected from itself.

The overall goal is to provide an efficient parallel optimization of the RFS algorithm. To achieve this, a shift in perspective of the problem domain requires revisiting. As RFS induces niching and speciation, it can be easy to get caught up with the overall work that the species performs. By employing a parallel RFS (pRFS) algorithm (Rogers, 2017), work is differentiated between what needs to be addressed by the species versus the cell.

With the new perspective, changes are proposed to better align RFS to the problem domain, without losing the generality offered by the original algorithm. Additional considerations are taken to better address parallel inefficiencies. In addressing this latter point, the concept of computational noise begins to develop.

Data often comes with noise, which requires empirical manipulation to reduce the potential for error. Similarly, noise, as it pertains to computation, is an important factor as well. This issue comes when subsystems of a model fail to capture the appropriate relationships

between specific elementary units. To expand upon this idea, this research addresses this in two areas: niche computation and initial proportion.

In both cases, the parameters used to define the work fail to encapsulate the pressure imposed by the conflicts on the corresponding species. In the case of the niche, it is important to ensure all species that impose pressure on a specific species are represented in the computation. This point is of minor importance as it arises from the need to save on complexity costs. With the use of pRFS, these limitations no longer need to be imposed on the algorithm.

More importantly, the role of initial proportion appears to be the predefined modeling of the conflict pressure. On a niche perspective, it's simply assigning a baseline fitness for all species in the population. Upon further research, this value seems significant in its ability to reflect the pressure that all species induce on each of the species.

For another perspective, consider the relationship between the initial proportion and the procedure for the niche computation. The initial proportion is where the algorithm is injected into the solution space; where the niche computation is how the algorithm searches through the space. Consequently, poorly modeling the injection point can significantly deteriorate the performance of the algorithm—leading to slow convergence in later generations if convergence is achieved at all.

1.1 Problem Statement

Resource-defined fitness sharing faces major scalability issues due to the inherent redundancies between the algorithm and problem domain. Further issues come from the implementation of a pRFS algorithm, which is heavily affected by imbalances intrinsic to the domain.

To address these issues, further refinement of the algorithm is performed for the reduction of both spatial and temporal complexities. Additionally, advanced parallel techniques are employed to address the imbalances found in the data.

1.2 Definition of Terms

Table 1. *Definition of Terms*

Term	Definition
Activation Matrix	A sparse matrix that represents the level of conflict all species have in the population, defined by the constraints of the problem.
c_{avg}	The average number of conflicting species for Sudoku puzzles at any size. When referenced for a particular puzzle, it is the exact number of conflicting species.
Constraint(s)	The rule(s) imposed to define an exact cover problem.
CUDA	Compute Unified Device Architecture
CUDA Dynamic Programming (CDP)	An extended model of the CUDA paradigm to exploit the variability in granularity of parallel processes.
(e)PRFS	Efficient Parallel Resource-Defined Fitness Sharing
Fitness Sharing	The use of merit as a limited resource.
Granularity	The measure of the amount of work performed by a task.
HPC	High-Performance Computing
Load Balancing (LB)	The process of partitioning and distributing a set of tasks over parallel processes in order to improve overall processing efficiency.
Niche	The set of features defined by an environment for the exploitation of biological organisms.
N	The number of elements that exist in a region of the Sudoku puzzle. $N = \lvert\{p_{size} \times p_{size}\}\rvert = p_{size}^2$
NGA(s)	Niche Genetic Algorithm(s)
(p)RFS	Parallel Resource-Defined Fitness Sharing
p_{grid}	The set of cells within a Sudoku grid. $p_{grid} = \{N \times N\}$
p_{size}	The size of a Sudoku puzzle, derived from the dimensions of the sub grid square. It is defined as the mapping of the puzzle's size to the natural numbers.
p_{space}	The set of cells within the problem space of Sudoku. $p_{space} = \{N \times N \times N\}$
RFS	Resource-Defined Fitness Sharing
Resource Sharing	The exploitation of explicit resources in a problem's domain.
Round Robin (RR)	A simple form of load balancing where processes or threads receive tasks in a cyclic fashion so to evenly distribute the work across all available processing units.
Species	A type or class of individuals that takes advantage of a particular niche.
S_{total}	The set of species which can form any puzzle in a particular size Sudoku grid. $S_{total} = p_{space}$
S_{active}	The set of species which form a population in a particular size Sudoku grid. The formal definition is derived to be. $S_{active} = \{c_{avg} \times N^2\}$

1.3 Description of the Remaining Chapters

The remaining chapters of this thesis are structured as follows. Chapter 2 discusses the methodology of the research, specifically laying out the broader questions that were used in guiding the phases of research and development. It concludes with the published algorithm description. A brief history of niching in GAs is reviewed, leading up to the development of RFS, in Chapter 3, which concludes with a look at how parallel computing and HPC techniques have been applied to the (N)GA domain.

Chapter 4 introduces the application of RFS for both open and real-world problems. Chapter 5 performs the complexity analysis of the RFS algorithm. A brief introduction of parallel computing with a dominant focus on the CUDA programming model is performed in Chapter 6. Chapter 7 introduces the parallel RFS algorithm, where Chapter 8 performs the analysis for each stage of development. Chapter 9 details topics for further research that failed to make it into this thesis. Finally, Chapter 10 provides the closing remarks of the thesis.

CHAPTER 2

METHODOLOGY

At its core, this thesis focuses on the following primary questions: What benefits can be achieved through the parallelization of the RFS algorithm? What performance can be achieved by this parallelization? What memory optimizations can be performed on RFS, and, by extension, what are the benefits? Conversely, what computational benefits can be achieved by ignoring memory constraints? How can parallel techniques, such as granular exploitation and load balancing, affect the parallel implementation? Finally, can the initial species proportion provide a buffer to the computational noise caused from conflict pressure? Subsequently, can altering the initial proportion improve the solution rate of the algorithm?

2.1 Algorithms and Data Structures

This research provides a comprehensive analysis of the RFS algorithm to elucidate the framework necessary for a highly efficient, massively parallel, (e)pRFS algorithm. The RFS algorithm is illustrated in Figure 4. In accomplishing this goal, several data structures are used in the implementation of pRFS. PRFS implements a one-to-one parallel replication of the serial RFS algorithm. Sudoku is used as a test bed, where problems can be generalized to any puzzle size. The primary data structure used is an array. The array is comprised of a defined species structure used for tracking evolutionary metrics. The remaining incidental memory requirements are managed as needed through the life of the algorithm. Because this was implemented using the Compute Unified Device Architecture (CUDA), the primary data structures are replicated in device memory.

PRFS with memory optimization refined memory management by allocating a puzzle structure. This structure holds all memory requirements needed to perform co-evolution with RFS. Two incident structures are kept. One is strictly device side memory used in the analysis

of the algorithm. The other holds device side memory allocations, such as incidental puzzle and device critical information. Additionally, it moves away from the three-dimensional array allocation in favor of a smaller array, reflecting the active number of species.

CHAPTER 3

LITERATURE REVIEW

A study on the resource-defined fitness sharing algorithm is performed for the extraction of inherently parallel structures. The analysis extends prior research on pRFS by developing an (e)PRFS algorithm. Specific focus is directed at the dichotomy presented by spatial and temporal optimization. Several phases of development are performed to balance the juxtaposition of space and time complexity optimization.

As RFS is an NGA, a historical review of niching in genetic algorithms is presented. Section 3.1 discusses the early research performed on niching. Section 3.2 introduces the primary research published on the RFS algorithm. Further considerations are taken in Section 3.3 on the use of parallel computation and high-performance computing (HPC) on NGAs.

3.1 History of Niching Algorithms

Evolutionary computation is a unified domain that evolved from the works of Lawrence Fogel, John Holland, Ingo Rechenberg, and Hans-Paul Schwefel. The domain is broken into evolutionary programming (Fogel et al., 1969), genetic algorithms (Holland, 1975), and evolution strategies (Rechenberg, 1973; Schwefel, 1974).

Niching is an advanced technique employed in the evolutionary algorithm domain (Goldberg, 1989). It is akin to theoretical ecology, agent-based artificial intelligence, etc., (Horn et al., 1994) and was derived from ecological niche theory by John Holland (Holland, 1975; Perry, 1984).

Niching is employed in multi-modal domains, which require multiple optima to be targeted. This searching mechanism is derived from the environmental concept of niches. Niches are the set of features defined by an environment for the exploitation of biological organisms (Holland, 1975). This definition was later refined by David Goldberg as the

organism's job or role in an environment (Goldberg, 1989). In both cases, organisms are the specific species suited for exploiting a particular niche.

While John Holland derived the theories behind niching algorithms, Cavicchio (1970) and De Jong (1975) were one of the first to introduce niche like behavior in genetic algorithm search (Goldberg, 1989). Cavicchio introduced preselection, which gave the offspring the ability to replace its least fit parent. De Jong's method, crowding, introduced an overlapping population. Offspring in this population would replace existing strings based on their similarities.

These two techniques provided the ability to maintain a more diverse population. Without either of these techniques, the population is subjected to uniform random selection. With the techniques, species are replaced faster, which reduces the overall payoff for each of the species.

In 1984, Zollie Perry provided the first direct application of niche theory to genetic algorithms (Goldberg, 1989). He formulated a framework for genetic adaptive plans by suggesting the use of tags in species and some of the schemata. The schemata are similarity templates developed by the simulation designer to characterize a species membership in a population.

By applying both tags and schemata, fitness would be imposed in specific niches. He went on to show an association existed between the niches and species defined by the tags. Consequently, this showed that incidental phenotypic properties offered non-uniform mate selection, which gives rise to distinct species (Perry, 1984).

In 1987 David Goldberg addressed concerns that GAs performed poorly when attempting to solve problems with multiple peaks—multimodal functions. He showed that, by incorporating sharing into GAs, roughly equal subpopulations of species would form around

the multiple peaks. Without sharing, the species would almost always converge on the highest peak (Goldberg et al., 1987).

One of the first references to parallel genetic algorithms was from Heinz Mühlenbein in 1989. He mentions the development of such an algorithm for running parallel computer systems. While parallel genetic algorithms can be run on multi-processing or distributive systems, they refer to those GAs, which incorporate multiple genetic algorithms to solve a single task (Cantú-Paz, 1998).

This is seen by the asynchronous parallel genetic algorithm ASPARAGOS, introduced by Mühlenbein (1989). The algorithm casts species to live on a 2D grid, where selection is performed locally. Then each individual performs local hill climbing with the use of a polysexual voting recombination operator. This operator is enforced on the local neighborhood of each species, where neighborhoods can overlap, and represents species mating and selection.

Specifically, ASPARAGOS introduces local search, with selection, and local hill climbing to genetic algorithms. In applying these changes, the algorithm was assessed on the quadratic assignment problem (QAP) and was shown capable of finding a new optimum for the largest published problem at the time (Mühlenbein, 1989).

In 1994 Jeffrey Horn showed niching was implicit in learning classifier systems (LCS), dissuading the argument that niching needs to be added. In doing so, he showed niching forced competing rules in the system(s) to share resources. Furthermore, a one-to-one correspondence was formed by mapping the LCS with implicit niching onto explicit fitness. That is, for each part of the LCS algorithm niching is used, it was up to the designer whether this niching was implicit or explicit.

Consequently, to enforce cooperation within LCS, niching is essential (Horn et al., 1994). This is because a cooperative set of rules is a diverse set of rules. Without maintaining this duality, GAs in LCS will fail to maintain diversity in the population.

Finally, the evaluation of sharing was shown to exhibit five characteristics. A) Sharing is ubiquitous. That is, what can generally be done with GAs can be performed with sharing. B) Sharing is fast, stable, and resilient. C) Sharing with selection alone performs complex, useful computations. D) Sharing supports the evolution of cooperation. E) Sharing can be modeled, predicted, and controlled (Horn, 1997).

3.2 Resource-Defined Fitness Sharing

Sharing has been shown to be a powerful paradigm to be exploited in GAs. The two techniques, resource and fitness sharing, have long been used to address problems in the multi-modal domain. Fitness sharing provides the ability to assess species within a population based on merit and was inspired by resource sharing for the simulation of function optimization (Horn, 2002).

In contrast, resource sharing allocates the available resources within a domain to induce niching. With niches formed in the population, the species would be rewarded by how much of the resources they acquired. This was shown particularly useful in learning classifier systems. However, the technique fails to resolve the complex non-linear dynamics achieved with fitness sharing.

In amalgamating these two techniques, the resources can be allocated explicitly as traditionally done in resource sharing, while the equilibrium calculations are kept simple to exploit shared fitness. The instance of this fusion is the resource-defined fitness sharing algorithm.

By combining these techniques, RFS was noted to be amenable to problems, such as nesting, tiling, layout, packing, and trim minimization (Horn, 2002). In extending the algorithm to these types of domains, it was assessed on the efficacy of nesting arbitrary, non-convex polygons in a substrate (Horn, 2005). Furthermore, rotation was included to allow shapes to be placed anywhere in the substrate at any angle of rotation.

1) INITIALIZE:

a) Generate initial set of species (unique chromosomes) S;

b) $\forall x, y \in S$: calculate pairwise intersection and store as $f_{x,y} \coloneqq \frac{|x \cap y|}{|x|}$

c) $\forall s \in S: p_s \coloneqq \frac{1}{|S|}$; // Uniform distribution across all species, initially.

2) LOOP: while (termination condition is false) do

a) $\forall x \in S$; Evaluate and store shared fitnesses as $f_{sh}(x) \coloneqq \left(\sum_{\forall y \in S} p_y * f_{x,y}\right)^{-1}$;

b) Calculate and store average shared fitness as $\overline{f_{sh}} \coloneqq \sum_{\forall x \in S}\left(p_x * f_{sh}(x)\right)$;

c) Calculate next generation species proportions p' as $\forall x \in S: p'_x \coloneqq p_x * \frac{f_{sh}(x)}{\overline{f_{sh}}}$;

d) Move to next generation by updating species proportions as $\forall x \in S: p_x \coloneqq p'_x$

Figure 1. *RFS Algorithm* (Horn, 2014)

In the experiment, a random initial population of 6,000 is used with the Gaussian mutation operator and no recombination. The operator adds a random value from Gaussian distribution to each species. A high mutation probability was used, along with binary tournament selection with continuous sharing (Horn, 2005).

With this starting population, the shared fitness of each species is evaluated. The fitnesses are then used to conduct M binary tournament selections. The winners in this selection are placed into the new population. Mutation is applied to the gene of each species in the new population. Finally, this new population replaces the current generation prior to looping on these procedures.

This procedure was assessed on two separate cases. The first evaluated the ability of RFS to nest an arbitrarily shaped polygon into another arbitrarily shaped polygon. The second experiment attempts to nest a convex polygon, such as a square, into non-convex substrates. Additional freedom was assessed with the use of rotation.

In both cases, RFS was shown capable of evolving species to cover most of the substrate. Consequently, the algorithm was shown capable in the general case of non-

convexity. Evidently, the algorithm can find cooperative solutions for identical shapes when the substrate is some arbitrary, non-convex, rotated polygon (Horn, 2005).

In extending the nesting research, three cases were used to assess the ability of RFS to perform tiling. Tiling is a form of the exact cover problem. The goal is to maximally place a shape onto a plane or substrate. In the latter case, the goal would be to minimize waste and maximize product output in manufacturing. For RFS, this translates to a maximally sized set of cooperating species. The algorithm was assessed on the performance of two species against one, two species against many, and many species against many.

In the first case, species A and B overlapped with each other in terms of resources. As species A and B maximally competed with a third species, C, it was found that C would become extinct after so many generations (Horn, 2007a).

In the second case, success was found, assuming specific criterion were met. First, the species must find an exact cover of the resources of all species. If this holds true, and the exact cover formed by the two species is the only exact cover, then they will dominate the population at the niching equilibrium (Horn, 2007b). This version of the algorithm was modified slightly to use proportionate selection as opposed to tournament selection (Horn, 2007b).

The final case addresses the competition of many against many. In this case, the population consists of two arbitrarily sized teams of species. If a team forms an exact cover, it will dominate the population at the niching equilibrium, forcing the other team to go extinct. This latter experiment showed the results to be more general than the original shape-nesting problem. Additionally, it shows the algorithm can solve the NP-complete problem of exact cover by k-sets (Horn, 2007b).

Since the evaluation of RFS is done on the Sudoku test bed, the remaining literature on the algorithm's ability to solve exact cover problems is used through this thesis. Briefly, a final

paper is used to show the evaluation of RFS against several commercial shape-nesting packages.

Limitations in the use of the commercial package was noted due to inexperience (Horn, 2010). The three algorithms incorporated in the package were ArtCam Insignia, ProNest, and OptiNest.

ArtCam is a computer numerical control machining software used in CAD machining packages. Upon review, this software package is now discontinued (Delcam Plc., n.d.).

ProNest is a nesting software package also available for CAD machining packages. ProNest was originally developed by MTC software but appears to now be distributed by Hypertherm (MTC Software., n.d.).

OptiNest is a general shape-nesting software offered by Boole (Boole and Partners, 2021), which appears to be used commonly in wood cutting and the furniture industry.

For the evaluation against these three software packages, a modified RFS algorithm is used—Pure Co-evolutionary Shape Nesting (PCSN) (Horn, 2010). PCSN uses the proportionate representation model, originally called the infinite population model. The model simulates infinite population through the application of proportionate selection. In this version of the algorithm, the species start with the same objective fitness. As co-evolution is performed, they reduce their shared fitness by competing for resources.

The results of this evaluation are represented in Figure 2. Specific attention should be given to the timing of the algorithms. While PCSN outperforms the three commercial packages, the time required to run the algorithm is exceptionally high at 70 seconds.

Using the default settings on the three packages, they were able to come close to the maximum number of circles nested within a few seconds. The best runs achieved up to an additional circle placed on the substrate, with a potentially significant increase in time to run.

Table 3: Results on Polygon P1.

Method or Package	No. of circles nested in P1	Approximate Run Time (seconds)
ArtCAM Insignia		
(with initial settings)	11	2
(best of all runs)	11	60
ProNest		
(with initial settings)	10	1
(best of all settings)	11	5
OptiNest		
(with default settings)	11	2
(with optimized settings)	**12**	**140**
Pure Co-evolutionary Shape Nesting (PCSN) (with diversity 10000)	**12**	**70**

Figure 2. *PCSN P1 Results* (Horn, 2010)

While PCSN outperformed the other algorithms, it does not pose the first instance of timing constraints faced by RFS. The original shape-nesting paper noted the running time required one to 40 hours on a 2 GHz Intel Pentium PC with 1.5 GB of memory (Horn, 2005).

Furthermore, none of the literature addresses spatial or time complexity, or parallelization, of the RFS algorithm. Instead, the research is heavily centered on deriving empirical evidence for the algorithm's use cases. As the evidence has shown, the algorithm has a competitive advantage, even over commercially available products, and would benefit greatly from the enhancements detailed later in this thesis. A comprehensive explanation of RFS as applied to Sudoku is given in Section 5.1.1. A list of potential areas to apply RFS is given in Section 4.2.

3.3 Parallel Niche Genetic Algorithms

As previously noted in Section 3.1, a distinction needs to be made when discussing parallel NGAs. This is due to inconsistencies in the wording of "parallel." Some individuals would use it in reference to those algorithms that can exploit parallel computers, while others would use the term to refer to those algorithms that use two or more genetic algorithms. In

either case, there is consensus that a parallel NGA is used to find multiple solutions simultaneously, such as RFS.

By this definition, RFS is considered a parallel NGA because it exploits niching, through both resource and fitness sharing, and finds multiple solutions. In the context of Sudoku, this may seem counterintuitive as only one solution can exist in a puzzle. However, every symbol found to exist in this solution is a sub-solution. The question then is whether RFS will find all the sub-solutions to a given puzzle.

As the parallelization of RFS is at the core of this thesis, it is important to understand how parallel computing and high-performance computing have affected the general class of niche genetic algorithms. Section 3.2 establishes that complexity optimization and the parallelization of the RFS algorithm have not been attempted.

Evolutionary algorithms have benefitted greatly from the parallel domain. Some of these algorithms take advantage of parallel processing techniques (Sato et al., 2011; Tsutsui & Collet, 2013). Other algorithms are used in load balancing (Zomaya & Teh, 2001; Kaliappan et al., 2016), and more have been used in high performance computing techniques (Dunlop et al., 2008). Furthermore, GPGPU processing has been a focus of research in the GA domain for quite some time (Sato et al., 2011; Wahib et al., 2011).

Upon review of the literature, NGAs have benefitted from the exploitation of parallel computation. Furthermore, they provide benefits in addressing HPC issues, such as through load balancing. However, this appears to be relatively one sided. For instance, using evolutionary computation techniques for load balancing networks and distributed systems have generated quite a bit of interest (Wang et al., 2014; Dam et al., 2015; Kaliappan et al., 2016).

Consequently, while NGAs have been parallelized quite extensively (Dunlop et al., 2008; Marco & Lanteri, 2000), they lack any research relating to their optimization through

load balancing. As pRFS is assessed for its initial viability to load balancing, the results declared in Section 8.4 show this research is warranted.

CHAPTER 4

APPLICABILITY

4.1 Open Problems

In general, the use of algorithms in open problems is a serious point of research in computer science. This opens broadly into attempts at answering the question, does $p = np$? This provides several complexity classes of problems: polynomial time (P), non-deterministic polynomial time (NP), complement of np (CO-NP), NP-hard, and NP-complete. See section 1.2 for the definitions to these types of problems. Consequently, much effort is placed in solving NP-complete problems. This extends into (N)GAs as well. Common problems that are used to assess the capabilities of these algorithms include the traveling salesman, New York City tunnels problem, Boolean Satisfiability Problem, the quadratic assignment problem mentioned earlier, and, of course, Sudoku.

Each of these problems pose serious challenges to test an algorithm's capability and have best seen results published regularly. As Sudoku is used in this research, an elaboration on the problem domain is given in the next section. There have been several different perspectives used in applying algorithms to Sudoku.

As is done in this thesis, Sudoku can be defined as a constraint problem (Simonis, 2005), a polynomial-time propositional satisfiability (SAT) problem (Lynce & Ouaknine, 2006), and, of course, the typical set covering problem (Horn, 2013). Similarly, many different methods have been employed in solving Sudoku. There is the GAuGE system (Nicolau & Ryan, 2006), Particle Swarm Optimization on CUDA (Monk et al., 2012), and co-evolution with RFS (Horn, 2014; Horn, 2017).

4.1.1 Sudoku

Sudoku is a Japanese combinatoric puzzle game that is played on a $N \times N$ grid. Clues are provided to the player and denote both symbol and location within the grid. The goal of the

game is to fill in the remaining symbols while respecting the rules and regions imposed on the grid. The regions of Sudoku consist of rows, columns, and squares. As these regions are superimposed on the grid, a fourth region naturally occurs: numerals. Each of these regions overlap in specific sub-grids across the puzzle with N copies of each.

The rules of Sudoku specify that regions contain one and only one of each symbol in the set $p_n = \{1,2,\dots,N\}$. Enforcing these rules and regions, coupled with the set of clues provided, ensures there is at most one solution for any puzzle grid. The diagram in Figure 3 details the regions of Sudoku.

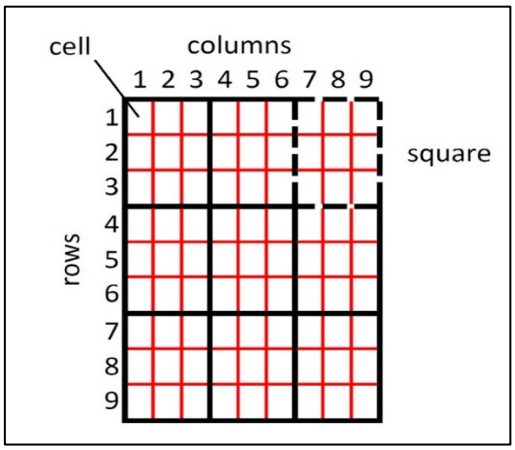


Figure 3. *Restrictions Enforced by Sudoku*

As the regions are placed on the grid to section off N cells, the shape of each region varies. The shape of these regions is straightforward as the rows and columns correspond to those in the grid. The square consists of a $\sqrt{N} \times \sqrt{N}$ sub-grid of cells, which partition the grid. The square regions begin at the top left corner of the grids and continue left to bottom.

The numeral region is particularly unique as it is the region with only one symbol. This doesn't contradict the previous statement as each cell starts with the full set, p_n, as potential candidates towards a solution. As the clues are fixed to the grid, the intersection of the set of clues and all copies of p_n across the overlapping regions is taken. Symbols in the intersection

set are removed from the grid to provide a set of viable symbols to be exploited for solving the puzzle.

The remainder of this section further abstracts the metrics of Sudoku. This is used to provide a clear definition of the problem for the application of RFS. The abstraction follows the definitions previously published in the work pertaining to the parallel RFS algorithm.

The typical Sudoku puzzle is played on a 9×9 grid. Conforming to the regions of the Sudoku, this results in 9 rows, columns, and squares—each of which contain 9 numerals. For this puzzle size, the square region is composed of a 3×3 sub-grid. This brings us to our first definition: puzzle size or level. As a puzzle level may also refer to its complexity, size will be the metric used to remove the ambiguity.

A mapping of the natural numbers is used to define the puzzle size: $p_{size} \coloneqq n | n \in \{\mathbb{N}\}$, which represents the size of the square region—$p_{size} \times p_{size}$. Extending this definition, the puzzle grid can be defined as the set $p_{grid} = \{p_{size}^2 \times p_{size}^2\} = \{N \times N\}$, where $N = n^2$. As each cell initially contains the set $\{1,2, \dots, N\}$, the problem space of Sudoku can be defined as $p_{space} = \{p_{grid} \times N\} = \{N^3\}$.

Recall that Sudoku is a combinatoric puzzle—so how do we represent the solution space for some arbitrary puzzle? To define this, we must use the information provided by the clues, which fixes the solution. The number of clues and their corresponding locations will determine the level of the puzzle. This is due to the way in which the clues remove conflicting symbols.

As each clue is placed, symbols conflicting with the clue are removed from a puzzle's p_{space}. Once the clues have been set, the remaining symbols form the puzzle. To gauge the level of the puzzle, one can compute the number of ways in which the remaining symbols can be placed into the grid. This can be done concisely by the combinatoric explosion computed by Eq. (1).

$$\binom{|clues \cap p_{space}|!}{(N^2 - clues)!} \quad (1)$$

To better visualize this explosion, Table 2 shows an estimation of the solution space for each of the puzzle sizes from 3 to 10 and 15. Even size 3 Sudoku puzzles can be particularly complex. So, increasing the size drastically adds to the difficulty in finding a solution. Clearly it becomes exceedingly difficult to efficiently find puzzles at the next higher level. Consequently, there were no available puzzles between the 11 and 14 size range. The only remedy to this would be to develop a Sudoku generator that can handle arbitrary puzzle sizes, which was unnecessary for the purpose of this thesis.

Table 2. *Average Solution Space for Each Puzzle Size*

Sudoku Size	Avg Conflicts	Estimate
3	$\binom{247!}{57!}$	5.34E+56
4	$\binom{829!}{154!}$	2.37E+171
5	$\binom{1970}{328!}$	4.28E+383
6	$\binom{5159!}{679!}$	5.97E+870
7	$\binom{7855}{1060}$	8.73E+1347
8	$\binom{15483}{1656}$	8.57E+2284
9	$\binom{23814!}{2549!}$	1.46E+3517
10	$\binom{39089!}{4000!}$	7.92E+5602
15	$\binom{103384!}{11823!}$	6.27E+15960

4.2 Real-World Applications

With the extensive research put into assessing the capabilities of RFS on set covering, any real-world extension of the problem would be applicable to RFS. Shape nesting on substrates was one problem predominately shown in the literature. This involves maximizing the number of a particular shape that can be cut out of a material.

Extending the enhancements provided by IpRFS to the shape-nesting domain would resolve the primary limitation of the original algorithm—the time required to find a solution. Consequently, the results of this research directly inspired Jeffrey Horn to pursue the MTRAC awarded project "Deep-Scale Evolution for Industrial Shape Nesting" (DSE) (Horn, 2022).

DSE is the extension of Horn's patented RFS shape-nesting algorithm. The targeted industries range from automotive manufacturing to textiles. These industries work with expensive materials and seek the best software for maximizing the product output, while limiting the waste produced in processing them. Furthermore, the project seeks to provide access to their high-performance computing (HPC) server(s) for entities to evaluate DSE on their own problem(s).

Additional areas in which IpRFS can be applied include the New York City tunnels problem, networking, and learning classifier systems, to name a few. The NYC tunnels problem is a common test bed for genetic algorithms. The goal is to set additional pipes of varying diameter in parallel to existing pipes to meet increasing demands.

Like the NYC tunnels problems, logistics can benefit greatly from IpRFS. Routing, labor, and supply are a few areas in logistics that could benefit. Routing, in particular, should see success. In addition to time management, the algorithm can take in real-world factors that directly impact route planning, such as weather, accidents, and other unforeseen delays. The HPC domain can equivalently use RFS for routing with load balancing as a common application of Gas in high-performance computing. Calculating how much labor is needed, how to distribute labor, or deriving capacity limits for existing and potential venues are all applicable areas for which RFS can provide solutions. This extends into the supply domain, as well, which is heavily interlinked with routing. Determining how much product needs to be made, where it needs to be shipped, and how much can be shipped in available containers are

all resource consuming endeavors. All of these require efficient algorithms to reduce time and cost, and RFS presents a highly adaptable solution to address them.

CHAPTER 5

RESOURCE-DEFINED FITNESS SHARING ANALYSIS

The procedure to implement RFS is conveniently simple and concise yet powerful. The primary focus is to cast the problem (domain) in terms of specific, finite niches to speciate the simplest units required for solving a corresponding problem. While Figure 4 illustrates the requirements to enforce the niching algorithm, it can be expanded further to incorporate the termination procedure required to assess the population for a valid solution. Two additional bullets suffice to accommodate this change: solution extraction and solution verification.

Note that these additions are for ease of use for the evaluation of RFS. That is, they are used to provide immediate termination criterion once a solution is found. Without this criterion, the algorithm can run for the predefined number of generations. If a solution is found within this generation limit, it will not be lost since the niche equilibrium has been achieved. Since the primary algorithm 4 is stated explicitly for applying co-evolution, it would suffice to add these bullets after the generation loop is completed.

For simplicity, a greedy approach, which enforces "survival of the fittest," is used for solution acquisition. The solution verification procedure is entirely problem dependent and is, thus, not expanded further. Below are the suggested procedure additions.

1. $\forall x, y \, \varepsilon \, S \, \& \, \forall c \, \varepsilon \, p_{grid} \, \& \, x, y \, \varepsilon \, c \mid f_{x,y}(c) \; = \; f(x,y) \; = \; max(prop_x, prop_y)$

2. Solution Verification

Figure 4. *Resource-Defined Fitness Sharing Algorithm Extension*

The remaining sections of this chapter address the spatial and temporal complexities of the algorithm. Further analysis of the algorithm is performed in the context of the problem domain assessed for this thesis—Sudoku. Section 5.1 details the respective complexities of the algorithm as presented in the literature (Horn, 2014). Section 5.1.1 continues this analysis in

terms of memory constraints. Unless specifically stated, the remaining chapters will refer to RFS complexity in terms of the Sudoku puzzle domain.

5.1 Algorithmic Complexity

The three procedures for RFS initialization consist of species generation, species extermination, and assigning of the initial proportion. The species generation is a casting of the smallest units in the constraints. The memory allocation will be addressed shortly but must be in place to differentiate the active species from the total population. To achieve this, the algorithm must iterate over the population of size S_{total}, which yields an upper bound of $O(S_{total})$.

The procedure in RFS 1.b takes as input a set of clues that are known to be part of the solution. A mechanism for removing the proportion of those species that directly overlap is used. This requires all species to be compared to each of the clues along all constraints, taking $O(|clues| \times constaints \times |S_{total}|)$ comparisons. Ignoring the weight of the constraints provides a complexity $O(|clues| \times |S_{total}|)$. This casts the population, S_{total}, to reflect the problem space of a specific problem as $|S_{active}| \mid p_{grid} < |S_{active}| < |S_{total}|$, where S_{active} is the active species.

The final procedure for initialization, RFS 1.c, Is an assignment to the proportion for each species. If a species was eliminated, its proportion is 0 and is assigned a uniform distribution derived by the accumulation of $|S_{active}|$. This requires two iterations over S_{total} to compute the number of active species and subsequently assign the appropriate value to the active species, yielding a complexity of $O(|S_{total}|)$. Looping is performed to transition the population from one generation to the next, while a solution has yet to be found. This is a sufficiently large, arbitrary number to provide the algorithm enough time to potentially find a solution.

Niching is performed in RFS 2.a. This weighs the level of overlap each species has with the rest of the population, which is then used to weigh the proportion of species. The subsequent product of weight and overlap are accumulated for corresponding species. This is performed for each of the constraints defined by the problem. Consequently, this results in a complexity of $O(4 \times |S_{total}| \times |S_{total}|)$.

This is mildly misleading as it is the computation for the activation matrix. If the population is held in memory in such a way as to exploit the constraints and structure of the domain, then it's more practical to iterate over each of the constraints to accumulate the proportions. This allows for those species without any overlap to be ignored. Consequently, a better complexity of $O(4 \times |S_{total}| \times N)$, where N is the size of each constraint, is achieved.

The population's average fitness is then computed in RFS 2.b. This computation is the dot product of the populations proportion vector and fitness vector. As the dot product is done on the number of species, the complexity yields $O(|S_{total}|)$.

The final two procedures, RFS 2.c and 2.d, apply the average fitness and shared fitness to compute the next generation of each species, then this value is assigned to derive the next generation, which is done in $O(|S_{total}|)$.

Solution acquisition, RFS 2.e, will depend on the criterion used for isolating species to form a potential solution. The current implementation assesses all species in each cell of p_{grid}. The symbol associated with the species with the highest proportion is used for the corresponding cell. This gives a complexity of $O(N)$. For solution validation, RFS 2.f, the procedure will depend on the problem being solved. The complexity assessment for this procedure appears in the next section.

5.1.1 RFS Sudoku Computational Complexity

This section provides an analysis of the RFS algorithm when applied to the concrete problem domain of Sudoku. As apparent from Section 5.1, the driving factor in computation is

the requirement to use a finite problem size that is sufficient to encompass any problem within a particular domain. Thus, S_{total} must be defined to gauge a more practical analysis. Recall Sudoku's problem domain is a subset of the generic domain n^d, where d is 6. This provides sufficient information to analyze how the algorithm will perform at each size, n, by casting $|S_{total}| = n^6 = N^3$.

Combining this value for S_{total} with the four constraints of Sudoku gives the complexity $O(|clues| \times 4 \times N^3)$. The number of clues, while variable, can be treated as a constant, which is due to the number of clues being static for any given puzzle. In addition, the average number of clues among puzzles is comparatively small, thus, dominated by the problem space of the domain. This would seem to fix complexity at $O(N^3)$ as most other procedures are assignments. The niche computation is the exception, which dominates the overall algorithm.

Recall that the niche is computed by masking the activation matrix. For computing the activation matrix, the complexity $O(|regions| \times |S_{total}| \times |S_{total}|) = O(4 \times \mathrm{N}^3 \times \mathrm{N}^3) = O(N^6)$ gives an upper bound. This isn't nearly as bad as it seems. Exploiting the constraints with respect to the structure of Sudoku allows this to be reduced to $O(4 \times \mathrm{N}^3 \times N) = O(N^4)$. This is better but clearly shows niche computation dominates the other procedures.

5.2 Memory Complexity

The memory complexity for RFS is bounded by the problem domain size, $|S_{total}|$. Generally, a minimum of $|S_{active}|$ species are required to cover problems in the domain. This number is derived from the enforcement of clues on the original S_{total} species. At minimum, the proportion and fitness of each species is maintained in memory. In addition, the average fitness is tracked. Finally, a solution space, of size $|\mathrm{p_{size}}|$, is used to extract, track, and verify a valid solution. Consequently, this yields the inequality $|\mathrm{p_{size}}| < |S_{active}| < |S_{total}|$. For

the purposes of this analysis and the refinement of the algorithm, the memory space of RFS is treated as $O(|S_{total}|)$.

Note that RFS computes the niche using a search mechanism in a way that the entire activation matrix is not held in memory. This will be important later in this thesis as the species conflict pairs are mapped and held in memory. In effect, holding the activation matrix in memory increases the bound to $O(|S_{total}| \times |S_{total}|)$. This is reducible to $O(|S_{active}| \times |S_{active}|)$, which will be discussed in Chapter 8.

5.2.1 Sudoku Memory Complexity

The RFS algorithm is bounded by the size of the problem domain, so it is sufficient to show the complexity with respect to Sudoku is $|S_{total}| = n^6 = O(N^3)$. Similarly, the grid size is $p_{grid} = (n^2)^2 = N^2$, which provides the lower bound for the number of species $\Omega(N^2)$. Succinctly, the memory space for RFS Sudoku varies along the inequality $N^2 < |S_{active}| < N^3$, where the active species, S_{active}, is dependent on the puzzle.

CHAPTER 6

PARALLEL COMPUTING AND TECHNIQUES

The use of parallel computing exploded at the turn of the millennia with the advent of multi-core, multi-threading hardware. The need for such equipment arises from Moore's Law, which shows diminishing returns and a limit to how small transistors can sustainably get. While many parallel computing models exist, CUDA was chosen for this research due to the massive parallelism it offers.

With the level of parallelism offered by CUDA, the RFS algorithm can be redefined to map threads to species. No longer did the algorithm need to iterate over the entire population, but merely needed to perform the work required of the species. In facilitating this adaptation, this chapter covers the basics of CUDA as it relates to this research.

The architecture and basic concept of CUDA are introduced in Section 6.1. This evolves in Section 6.2 with a brief description of the benefits provided by the extension of CDP. Section 6.3 offers an explanation on hardware concurrency, then briefly covers load balancing.

6.1 Compute Unified Device Architecture (CUDA) Model

NVIDIA provides massive scalability solutions through its CUDA enabled hardware and the corresponding toolkit. The current toolkit is 11.6.1. While it best aligns with devices at compute capability 8.6, it provides backwards compatibility for earlier compute devices. How does the model work though? NVIDIA designed the CUDA toolkit to be amenable to any programming language. Figure 5 shows a simple diagram illustrating how new languages can incorporate the toolkit. All research in this thesis was performed using the CUDA wrapper for C/C++.

The wrapper extends languages to provide for additional syntaxes, lambdas, and allocators, which are used to exploit the underlying hardware. The CUDA-enabled device requires memory allocation to use. Both host and device allocate memory as required using the

cudaMallocHost and cudaMalloc allocators. The allocation mechanism performs the same operation as vanilla malloc.

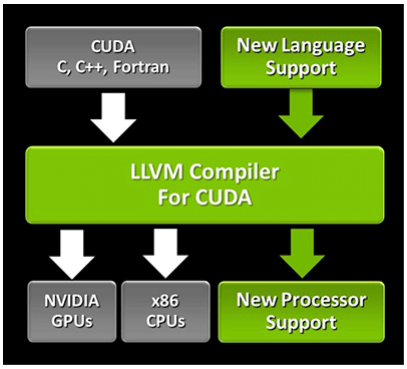


Figure 5. *CUDA LLVM Compiler Structure (NVIDIA et al., 2020)*

The allocators have different purposes. Host memory has all allocation mechanisms present in C/C++ with the additional wrapper—any form of memory allocation will suffice. The CUDA allocator provides additional benefits for memory transfers as it pins memory resulting in page-lock for the entire memory address range. On the device side, the allocator initializes a pointer to global memory. The memory hierarchy is shown in Figure 6b. While CUDA offers several different forms of memory, this thesis focuses on the use of global, shared, local, and registered memory.

Global memory is the large memory space on the device that is akin to host side RAM. Shared memory is faster and much smaller but provided by each of the streaming-multiprocessors (SMs). This memory is exploitable by thread blocks launched via CUDA kernels. Below shared memory is register memory, which is the fastest on the device. Finally, there is local memory, which is reserved for each thread and is a small subsection of global memory.

Several comments about CUDA kernels have been made by now, so what are they? A CUDA kernel is a device function that is executed by the device and is the injection point for utilizing the device hardware. The host calls the kernel, which shifts control to the device. The

work defined in the kernel is then processed. Once all work has been completed, the instruction pointer is returned to the host.

Kernels are defined using the $\_\_global\_\_$ lambda, which identifies the function as a device kernel. The kernel is launched with the syntax $myFunction <<< blockCount, threadCount, sharedMem, stream >>> (...)$. We will go over each of these parameters momentarily.

Like kernels, device functions are denoted by the device lambda. Functions declared in this manner indicate to the kernel a function is usable strictly by the device. Contrarily, host is used to denote purely host-related functions. It is also possible to combine these lambdas, $\_\_host\_\_ \_\_device\_\_$, to reuse procedures from either host or device. This later point comes with the caveat as the function must be written explicitly for the host. No device code, such as $threadIdx.x$, can be used.

This brings us to thread execution. The diagram in Figure 6a illustrates the execution of threads on the device. Threads are launched via the kernel by providing a thread definition composed of the number of blocks and threads per block to use: $blockCount$ and $threadCount$. Technically, threads per block can range from $1 \leq threadCount \leq 1,024$. CUDA best practice dictates multiples of 32 are used to conform threads to warp sizes. A warp is a set of 32 threads, which execute the same instructions.

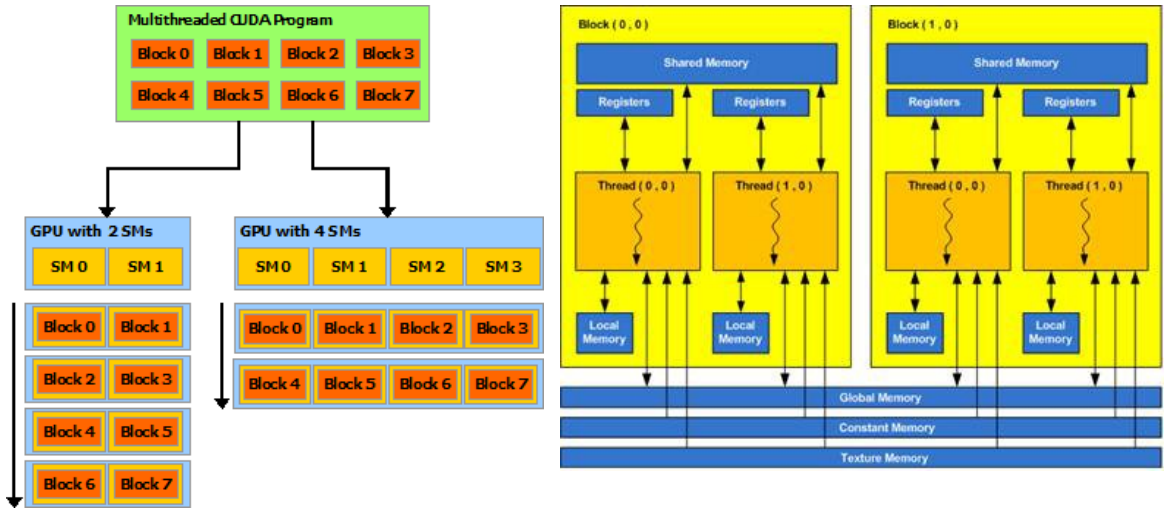
Both $blockCount$ and $threadCount$ parameters can receive a single integer value. However, this value is implicitly cast into a $dim3$ structure. The explicit definition is $dim3(x, y, z)$. The coordinates are respectively used to define threads in three dimensions. In this manner, a kernel can achieve up to six degrees of freedom, assuming the threads per block does not exceed the 1,024 threads.

Upon execution, the thread definition defines the grid in which the threads will operate over. The grid is composed of blocks and threads. Corresponding thread structures are built

into the CUDA API. The primary structures are $threadIdx$, $blockIdx$, $blockDim$, and each of them contain an x, y, and z variable. Mapping threads is simple, and a typical use is provided in Eq (2).

$$tid = threadIdx.x + blockIdx.x * blockDim.x \tag{2}$$

The other two parameters of the kernel launch are shared memory and streams. Shared memory, or dynamic shared memory, provides the ability to allocate a small amount of fast, block specific memory. The maximum amount varies between devices but has remained steady at 46KB for most desktop GPUs. However, research-oriented devices are configurable up to 164KBs, as of the ampere architecture.



(a) CUDA threading model (b) CUDA memory architecture

Figure 6. *CUDA Thread and Memory (Romero & Urra, 2022)*

The allocated shared memory is defined for each block of threads. Consequently, all threads within a block can access the same shared memory space, but no threads out of the block can access data in this space. For operations that require communications between blocks, partial computations can be done on the shared memory, resulting in each block being written back to global memory. This is commonly seen in parallel reductions.

Finally, the stream parameter defaults to 0, which indicates the primary CUDA stream will be used for kernel execution. A CUDA stream can be allocated to run the kernel on an explicit stream. Kernels executed on explicit streams can enjoy concurrent execution with the default or other explicit streams, though concurrency is not guaranteed.

6.2 CUDA Dynamic Parallelism

CUDA dynamic parallelism extends the CUDA model to address varying granularity. Granularity is the measure of how much work is needed to perform a task. There are three levels of granularity: fine-grained, medium-grained, and coarse-grained. Work that can be broken into many small tasks falls under fine-grained parallelism. Medium-grained parallelism has tasks that are larger than fine-grain but smaller than coarse-grain. Work that is split into large tasks falls under coarse-grained parallelism. In general, granularity balances task size with computation time.

In the CDP model, kernels are provided the ability to launch children kernels. These kernels have their own thread definition, which is only limited by the hardware. Subsequently, a procedure that tackles a broad problem can be deployed as a coarse-grained task. As the threads complete work, areas of the problem requiring more processing can be refined by the deployment of children kernels. Once this dense pocket of work is completed, the child kernel returns instruction back to the thread of the parent kernel.

With the ability to launch children kernels, recursion becomes an additional benefit imbued in CDP. However, this does come with some caveats since there is a hardware-imposed limit, which restricts recursion to a depth of 24. However, memory constraints will more likely become an issue before reaching this depth.

6.3 Load Balancing

A common issue in parallel computing is the effective parallelization of tasks and computations. This boils down to the work done by the processor with the number of threads defined to complete the work. Issues occur when the amount of work assigned to threads becomes imbalanced. When thread work is unbalanced, varying levels of computational time may be lost. The greater degree of imbalance, the more loss, and subsequently more time is required for an algorithm to run.

Generally, work done by the processor is partitioned into highly variable, difficult to estimate size pieces (Kumar et al., 1994). To address the variability of work available to the processor, load balancing techniques were introduced. Load balancing schemes offer various ways to allocate more work to idle threads from other threads that contain too much. In effect, they increase the utilization of the processor(s) by reducing the inefficiencies caused by computation imbalances. These imbalances can be due to scheduling, data, etc. The use of load balancing in this research is in the early phases. The results claimed were acquired using the round robin (RR) scheme. In this scheme, once a thread has completed its work, it checks the next thread for more work. If there is no work, the first thread dies. An extension of this is asynchronous round robin (ARR) load balancing, where the thread will not die but continues looking at each subsequent thread until it finds work to do.

As a “global” version of asynchronous round robin, there is global round robin (GRR). In this scheme, there is global a target indicating the next thread to be selected for the transfer of work. When a thread completes its task, it acquires the index held in target, then updates the value to the next thread. The thread then attempts to acquire more work from the thread assigned as indicated by the target variable. This will occur for all threads until all work is completed.

Nearest neighbor (NN) load balancing (Kumar et al., 1994) works similarly to round robin. When a thread runs out of work, it sends a message in a round robin fashion requesting work from its immediate neighbors. This scheme is similar to ARR but limits the threads available from which to receive work. ARR, on the other hand, has each thread use an independent target variable that cycles over the thread pool.

CHAPTER 7

PARALLELIZATION OF RESOURCE-DEFINED FITNESS SHARING

As shown in the analysis of RFS, the spatial and temporal complexity provide significant limitations to the algorithm. Most notable is the domination of niche computation at $O(N^4)$. Memory is generally manageable but does pose scalability issues if the entire $O(N^3)$ species population is retained.

In this chapter, we will look at how to reduce the spatial complexity of RFS, while effectively managing temporal complexity. Section 7.1 begins with the analysis of key parallel structures inherent between the RFS algorithm and Sudoku. These structures are then exploited to provide a naïve, one-to-one, performance baseline between the serial RFS and CUDA RFS algorithms.

The baseline parallel implementation is extended in Section 7.2 by addressing spatial limitations in the parallel algorithm. To accomplish this, inefficiencies are identified, and solutions provided to resolve them. Finally, Section 7.3 exploits advanced parallel techniques to improve overall performance. The section begins with a granular assessment of the algorithm, which is used to apply CDP. Load balancing, isolation of conflict pairs, and a noise assessment on initial proportion is conducted in Section 7.3.2.

7.1 CUDA Resource-Defined Fitness Sharing

In using GPGPUs, algorithms can exploit massively parallel code to make more efficient use of multi-threading hardware. RFS is one such algorithm, as it revolves around speciation of the most basic units of a problem. The speciated units then accumulate and manipulate the sharing of parameters to propagate the population towards a solution. So how will this work?

Memory must first be allocated on the device. Utilizing CUDA's memory allocation wrapper, we allocate N^3 species on both the host computer and the device. The initial implementation is done using a 3D array of species, which is later converted to a 1D array. This transition better conforms to CUDA best practices due to the linear memory arrangement imposed on the device. For the implementation of parallel RFS, a host copy of the population is generally unnecessary—all computations can be done on the device. However, debugging, metric acquisition, and metric analysis is easier to manage with a host side copy of the population.

With memory allocated, the population must be molded to represent the puzzle. The molding process happens by eliminating those species that are in direct conflict with the clues. The elimination procedure happens by mapping a thread to each of the species. By utilizing this mapping, each species can assess whether it should be exterminated or not.

To achieve this kind of mapping, the initialization procedure uses a dim3 structure for defining block and thread counts. Since this structure can define threads along the x, y, and z axes, the thread and block definitions are defined by $dim3(n, n, n)$. Using the prescribed thread definition, kernels can launch the exact number of threads required to map to each species.

While the use of the $dim3$ structure provided an effective means to implement pRFS, it immediately restricted the algorithm's scalability due to hardware-imposed limits. The maximum number of threads that can be launched is 1,024 per block. With the dim3 thread definition, RFS can only tackle up to size 10 puzzles. This is reflected in the results section, as timing for the size 15 puzzle was not achievable, and a simulated value had to be used.

Two approaches were used in the implementation of the initialization kernel. The most direct method launches the threads, as defined above, where each iterates over the clues list. If the thread finds a clue with which it conflicts in any region, it exterminates itself.

The other method creates a CUDA stream for each clue. The extermination kernel is assigned a clue and stream and is then executed asynchronously. Now each clue has been parallelized to remove all species that conflict with it.

For this implementation, the population is copied back to the host to compute $|S_{active}|$. Once computed, another kernel is launched to assign the initial proportion, $(|S_{active}|)^{-1}$, to the active species. The remaining species are labelled as "exterminated," and the proportion is assigned 0.

The remaining memory required for executing RFS is now allocated on the device. This most notably includes a solution array, a double for tracking average fitness, a Boolean flag to indicate solution status, a proportion array, and a fitness array. The duplication of memory from the latter two allocations will be explained momentarily.

With the allocations completed, the algorithm enters the looping phase, which begins by updating the block and thread values to the $|S_{total}|$ definition. The niche computation kernel is then launched. A thread is mapped to each species in the population, which then iterates over each of the regions. The proportions of all N species are accumulated and weighed appropriately. The final value is then added to the proportion of the species associated with the thread.

The accumulated proportions are used in the next kernel to apply a proportionate selection. The proportion and shared fitness for each species are subsequently copied to the respective arrays. This is done to use the dot product procedure in the CUBLAS library, which is why there were duplicate memory allocations. With all values computed, the next generation is derived and assigned to the respective species.

Now the thread definition is redefined by $|p_{grid}|$ to perform the extraction and solution verification kernels. For extraction, each cell in p_{grid} is assigned a thread to iterate over the

numerals associated with the cell's location. Each species is assessed to find the best fit based on proportion. The numeral associated with the selected species is used as the cell's solution.

Using the solution verification kernel, each thread allocates an array of size N to check for duplicate values. If a duplicate value is found, the solved flag is set to false—forcing all threads to terminate on their next iteration. Upon completion of the kernel, the solved flag is copied back to the host. The host's flag is used to indicate whether the algorithm should continue. If the flag indicates a continuation, the loop will do so for the indicated number of generations.

7.2 Memory

This section details two specific goals for the refinement of the CUDA RFS algorithm: the reduction of memory required for processing and the exploitation of the CUDA memory hierarchy. As it was shown in Chapter 5, the memory requirements of RFS Sudoku are bounded by the inequality $|p_{grid}| < |S_{active}| < |p_{space}|$. In view of the upper bound on S_{active}, hardware limitations would not be as restrictive on the algorithm's scalability.

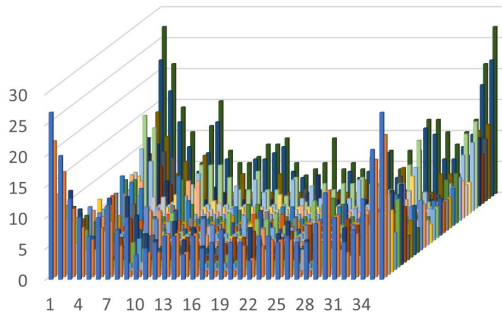


Figure 7. *Species Distribution of Size 6 Sudoku*

The reduction in memory provides more benefits than pure scalability. Recall that each clue used to define a puzzle removes directly conflicting symbols from the problem

space. This turns out to be quite significant, especially as the puzzle size increases. Figure 7 shows the p_{space} of a size 6 puzzle after conflicting symbols have been removed.

As can be seen, a large portion of the initial p_{space} has been removed. While the overall reduction is more complicated, a simple bound on the number of symbols removed by a clue is computed by Eq. (3). The complication in deriving a more concrete formula comes from the subsequent application of clues. The first clue will remove the corresponding number of symbols defined by Eq. (3). However, each subsequent clue application will remove less than the previous.

$$4N - 2\sqrt{N} - 1 \tag{3}$$

The reduction in memory also improves the utilization rate of each warp deployed. Figure 7 provides a nice visualization for the puzzle's shape, but Figure 8 provides the average estimate of this reduction for each puzzle size. The improvement in warp utilization should be obvious by now, since most threads, and subsequently warps, perform little to no useful work.

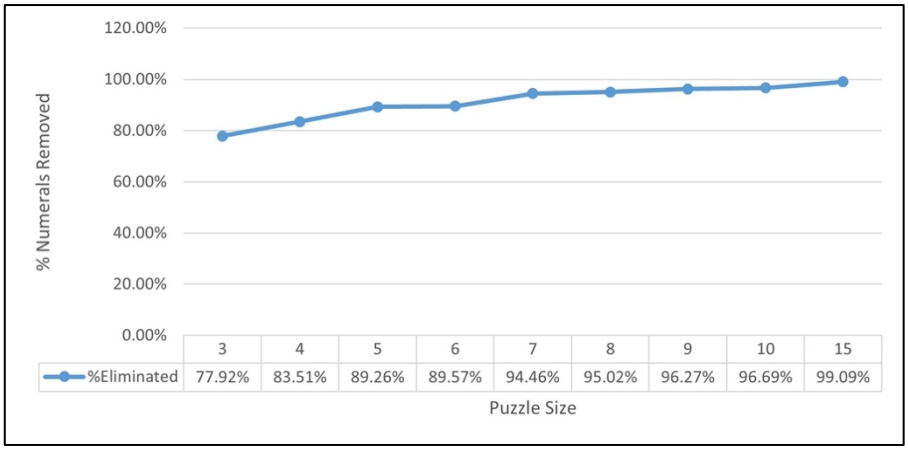


Figure 8. *Average Numerals Removed*

For this implementation, S_{total} is reduced to S_{active} without retaining the structure of the puzzle. Instead, each species tracks its location by mapping its 3D index to the corresponding row, col, numeral, and square. To accommodate this change, thread launch

configuration is dynamically defined by the size of S_{active}. Eqs. (5) and (4) provide the computations for deriving block and thread count.

For $baseThreadCount$, the value is kept at 64 for the purposes of analysis. Likewise, the maximum threads per block, 512, was determined to be the most efficient. The values between $512 < threads \leq 1024$ showed no further improvement. In practice, the number of threads would often degrade performance as it approached 1,024, which is due to the small puzzle sizes. Providing the dynamic thread definition ensures that enough threads are launched for each puzzle and puzzle size. Additionally, it limits the upper bound of threads, so only one SM is underutilized.

$$threads = min\left(\left(\left(\frac{Sactive}{512}\right) + 1\right) * baseThreadCount, 512\right) \quad (4)$$

$$blocks = \left(\frac{Sactive}{threads}\right) + 1 \quad (5)$$

The importance of the CUDA memory hierarchy was detailed in Chapter 6. To effectively implement the pRFS algorithm, the hierarchy was closely considered. Dynamic shared memory was tested for all kernels but with limited success. Only the kernels that perform reductions benefited from the exploitation of shared memory. These are the dot product and add reduction kernels. Locality and the limited amount of shared memory are the driving factors as to why the remaining kernels did not benefit.

The niche computation kernel failed to benefit from shared memory due to locality. The time required to load species data into shared memory and then exploit it performed poorly against using register memory. Furthermore, there was no reuse of the species once the data had been acquired. Section 9.1.2 presents a new approach to the niche computation, which would benefit from shared memory.

Section 8.3 addresses the work and granularity issue in more detail. The arrays allocated in the solution validation were replaced with dynamic shared memory but hindered the performance. Instead, the change proposed at the end of this section was used.

Now let's look at the remaining changes and how the implementation varies from the original CUDA RFS algorithm. Due to the heavily interlinked memory optimization in the CDP implementation, the initialization procedure is deferred to Section 8.3. The thread configuration is implemented as described for those kernels requiring $|S_{active}|$ threads. The latter two kernels' thread count is redefined using $|p_{grid}|$. There are three main changes to consider for the niche computation kernel. First, gating is used to ensure only $|S_{active}|$ threads are used to compute the niche count. This is important because the kernel will launch more threads than are needed. The final block will be the only one that is underutilized. The degree of this limited utilization will depend on S_{active} and the number of threads per block.

The next change is an accommodation for the reduction of population to S_{active}. The memory reduction effectively destroyed the structure exploited in the first implementation. Consequently, each species must compare itself to all other species, which effectively computes an $|S_{active}| \times |S_{active}|$ dense activation matrix.

In deriving the mapping to assess region conflict, it was noted that up to N species would ever be in conflict for a given region. This value is used to assess species that directly overlap (J. Horn, personal communication, March 3, 2022). In only selecting the N conflicting species, up to $N^2 - N$ conflicts are ignored. This change shifts the focus of overlap to the cells versus between those of the same symbol.

With the reduction in computational complexity provided by pRFS, this limit no longer needs to be enforced. Now each species is compared for the N copies of the region where it exists. All species that fall into this N^2 grid are accumulated, and the weighted product is derived once all comparisons are done.

To put this into perspective, consider the species [3, 4, 5] as the row region is evaluated. Originally, the x value would be free, locking down on y and z. All species in [x, 4, 5] would conflict with [3, 4, 5] and have their weighted proportions added to 3. The proposed change has all species in [3, x, y] in conflict with [3, 4, 5].

The number of conflicts is accumulated, and the weighted product is derived once all comparisons are done. An integer is used to accumulate the weight, which provides computational savings over the floating-point arithmetic. Finally, the niche computation and proportionate selection kernels are merged to reduce the total number of kernels.

The remaining changes to the algorithm are mostly minimal. The dot product and next generation kernels are modified to accommodate the reduced memory. A dot product reduction kernel was tested to attempt improvements over the CUBLAS implementation, which showed equivalent performance. The solution acquisition and verification kernels were converted to device functions and consolidated in a check solution kernel.

Finally, solution verification was overhauled to remove the memory allocation for checking the conflicting symbols. Instead, each thread iterates over a double-nested loop for a total of N iterations. The two iterators are combined to check the corresponding locations in each region. If any symbol is found to conflict with the symbol in the cell, the solved flag is set to false, and all threads stop at their next iteration. The result of this generation is copied back to the host, which indicates whether the algorithm should continue.

7.3 Parallel Resource-Defined Fitness Sharing

This section details the amalgamation of various techniques to better understand and exploit the parallel RFS algorithm and begins by addressing the effects that memory modification had on the initialization procedures. CDP is then used to exploit granular parallelism within the kernels. From there a greedy memory approach is implemented to assess the performance of RFS if conflicting key pairs are held in memory. Round robin load

balancing is then used to normalize the amount of work each thread does for the niche computation. Finally, we explore the effects of noise in the algorithm by varying the value of initial proportion.

The goal in the previous section was to show how the memory utilization can be reduced to the S_{active} species. However, the derivation of this set for the population depends on $|p_{space}|$. This implies that the upper bound on memory will be reached at some point in the algorithm. So, why not use it immediately, then reduce the memory space to the S_{active} species?

7.3.1 CUDA Dynamic Programming

The larger memory space was used to facilitate the extermination of species. For this goal, the extermination was not done in the context of species but as floating-point labels. Additionally, the host side parallelization of the clue list was removed. Instead, CDP is used to parallelize the clues, which would launch a child kernel to exterminate the species.

The exterminate kernel's launch configuration is changed to use $|p_{grid}|$ for the thread definition. Once the kernel has been deployed, each thread creates and initializes a unique stream, which reduces the likelihood it would be forced to serialize with the other threads. The threads then assign a clue to the child kernel, which is launched with the same thread definition.

Upon execution, the child kernel will gate N threads for use in the extermination process. The regions of the clue are derived to fix the corresponding regions in 3D space. The thread uses its ID in the free region, and the remaining regions of the clues are fixed. This directly maps the thread to four conflicting symbols, which removes the need to search for them. The index is used to set the value for the conflicting symbols to 1.

As only N threads are used, there is significant waste in the thread definition. This waste can be reduced by increasing the gate limit to $4N - 2\sqrt{N} - 1$ threads, which is the

total number of conflicts possible. Using this value in the thread definition would further reduce the waste generated for the kernel(s).

Initialization returns to the host, which proceeds to compute $|S_{active}|$. Using the floating-point values in the previous kernel presents an additional benefit. The CUBLAS' dot product kernel can now be used for computing the total number of removed symbols.

The value returned from the dot product is used to compute $|S_{active}| = |S_{total}| - p_{sum}$, where p_{sum} is the sum of the number of clues and conflicting symbols. Now $|S_{active}|$ is computed, and the population array can be generated. Finally, a kernel is deployed to initialize each species with the necessary information to perform the algorithm. The thread definition at this point has been changed to $|S_{active}|$.

Since the Sudoku structure was lost in the memory reduction, each species must now search the entire population to find its conflicts. In effect, the $|S_{active}| \times |S_{active}|$ condensed activation matrix must now be computed. CDP is used as an attempt to alleviate the computational cost that the memory reduction imposed. To convert niche computation to a parent kernel, the $|S_{active}|$ thread definition is used. Each thread is gated for the $|S_{active}|$ population, and children kernels are launched with the same thread definition. In addition, the child kernel carries over the identifier of the species corresponding to the niche computation. Each thread then compares the population to the host species to find the conflicts. The conflicting species will atomically add their weighted proportion to the host species proportion.

The dot product and next generation procedures do not have granular considerations to exploit CDP. This leaves the solution acquisition and solution verification. While these are discussed here, their performance is not assessed in the results. The explanation for this is addressed in Chapter 8.

The solution acquisition kernel must search the population to find the best fit species for each cell. CDP is attempted to remove the search time. The parent kernel is launched with

the same $|p_{grid}|$ thread configuration as in the previous version. Each thread then launches a child kernel with the $|S_{active}|$ thread definition to parallelize the loop. An atomic max function is used to make the appropriate comparison. However, it might have been more efficient to do a max reduction.

The assessment of CDP on the solution validation was attempted prior to the memory optimization presented in the previous section. For this version of the validation, shared memory was utilized in the child kernels, which replaces the allocation of global memory. The child kernel is launched with the same thread definition as the parent. Once executed, $4N - 2\sqrt{N} - 1$ threads were gated for use. The threads then assessed all symbols in the four regions of the parent cell. Shared memory is used to track duplicate symbols, and the solved flag is set if a conflict occurs.

7.3.2 Greedy Memory and Load Balancing

Memory optimization was explored to ensure GPGPUs scalability. How does the opposite side of this dichotomy work though? Instead of reducing memory, use it to reduce the number of operations needed. A lower bound on the species array was shown to be concretely set at $\Omega(|S_{active}|)$. The upper bound is then set to $O(|S_{active}| \times |S_{active}|)$, which defines the size of the sparse activation matrix of the puzzle. For the niche computation, each thread searches the population for conflicting clues. This process masks the activation matrix so that, instead of doing the search, it holds the conflicts in memory. However, holding the entire matrix would be wasteful due to its sparsity.

Instead, we will separate the conflicts from the activation matrix by mapping them to conflict key pairs. This immediately saves a vast amount of memory since most species do not conflict. While comparatively small, this also provides the ability to remove the location information required to implement niching. As this information is used only to derive the key pairs, it can be traded for each species location into the population array.

In addition to storing the conflict key pairs, round robin load balancing is implemented to assess whether further investigation is worthwhile. This form of load balancing has each thread check the subsequent thread(s) for additional work. To acquire the conflict key pairs, the search performed in niche computation is extracted and used in the initialization stage. Instead of weighing proportions, the procedure counts how many conflicts each species has.

With the number of conflicts known for each species, an add reduction is used to sum up the total number of conflicts in the population. This procedure also employs max reduction and min reduction to acquire the range of conflicts in the population. Once completed, memory can be assigned to hold each conflict key pair. In addition to holding the respective species, the conflicts weight is also held—reducing the number of key pairs that need to be stored.

A kernel is then launched to find and store the conflict key pairs. The thread configuration is defined by the number of conflicts. Before the threads can start assigning key pairs to memory, they need to compute where in memory the conflicts will be stored. To do this, the thread adds the conflict sums of the species that come before it in the array. Then the thread proceeds to generate and save the conflict key pairs associated with its species.

With the conflict key pair list, the niche computation kernel is launched using the same thread configuration as the conflict sum kernel. The threads are gated to the total number of conflicts. The indices of both species are then pulled into register memory. The weighted proportions are computed in register memory prior to being atomically added to the host species proportion in global memory.

For load balancing, the kernel will use the $|S_{active}|$ thread definition. The kernel implementation will now incorporate a loop, allowing the threads to iterate over the conflict pairs. Once a pairwise computation has been completed, the thread ID is increased by the value used in the thread definition—the value of which is held as $gridDim.x * blockDim.x$. The looping occurs until the thread ID increments beyond the limit set by the number of conflicts.

7.4 Noise

The solution rate of RFS stands at 94% (Horn, 2014). While this is good, it is not 100%. One thought to increase the solution rate would be to exploit a commonly used technique for solving Sudoku puzzles with the removal of trivial solutions. Often, one or more numerals remain after the extermination process, but they have no conflicts. Since they have no conflicts, they are guaranteed to be part of the solution—so why not fix them to the solution?

This idea builds into a gradient ascent theory to investigate. This will be discussed in Section 9.2.1. Implementing this idea detracts from the evaluation of RFS. So, how can we attempt to represent the idea without drastically changing the RFS algorithm?

Let us consider the noise within the S_{active} species by addressing the initial proportion. Recall that the initial proportion is a uniform distribution over the species to ensure that the assigned value falls between 0 and 1. While the uniform distribution provides a means to encapsulate the proportion with respect to the population, it ignores the pressure placed on each species by the other species in its constraints. Subsequently, the distribution adds too much weight from all species not within its constraints.

So, let us modify the initial proportion in a manner to better reflect these constraints. Four settings of the initial proportion were tried. The first trial sets each species' initial proportion to 1, which allows conflict pressure to be the primary driving factor. The second trial sets the initial proportion to $(sc)^{-1}$, where sc is the total number of conflicts for the species, to see the effects of conflict weight. The third trial takes the opposing probability of $1 - (sc)^{-1}$. Finally, the last trial defines proportion in terms of p_{space} and the number of conflicts: $\frac{|p_{space}|\%sc+1}{sc}$. This retains the variation of the previous two trials by assessing the context of the species based on its p_{space}.

CHAPTER 8

RESULTS

This chapter goes through the changes proposed to the RFS algorithm and performs a side-by-side comparison of the spatial and temporal complexities they enforce. Performance data will be reviewed after the complexity analysis to better understand how these changes affect the algorithm. By comparing the complexities to the data, the results can be better assessed and validated to ensure they conform to the expected outcome.

With this step-by-step analysis, the end goal of the chapter is to provide a comprehensive understanding of what performance enhancements can be achieved with parallel RFS. The subsequent analysis lays the foundation for future research for implementing a highly efficient parallel RFS ([e]PRFS) algorithm.

The GPGPU used to normalize the results for this thesis was the RTX 3090, which is equipped with 82 SMs. As this card has compute capability 8.6, each SM supports a maximum of 16 active thread blocks and 64 active warps (NVIDIA et al., 2020). The utilization rate of SMs is defined by the ratio of active warps to the maximum supported active warps. So, to achieve maximum utilization, each SM needs a minimum of $64 * 32 = 128$ threads per block. Putting this together, the device can handle up to $82 * 16 * 32 = 41{,}984$ threads in parallel.

8.1 CUDA RFS

The use of CUDA for transitioning RFS to a parallel algorithm has easy and immediate consequence threads that can be assigned for every numeral in a problem's p_{space}. As each function in the RFS algorithm can be defined by a CUDA kernel, we can map complexity to the work performed by each thread, which removes the need to iterate over the three nested loops required to process the 3D space of the puzzles.

We were able to derive an $O(4 \times |clues| \times N^3)$ complexity for the serial extermination method. The proposed implemented parallel change had the host launch CUDA

streams for each of the clues, which, in effect, provides the ability to process all clues in parallel. Keep in mind that there is no guarantee the streams will execute concurrently. As stream execution is managed by the device controller, the best-case scenario is the concurrent extermination of all clues.

This reduces our extermination complexity to $O(4 \times N^3)$ for each host stream launched. The exterminate kernel is then defined with at minimum $|p_{space}|$ threads. Each thread is responsible for evaluating whether the associated symbol conflicts with the specific clue. If it does, the value is set to 0; otherwise, the species remains unchanged. This provides the thread complexity of $O(1)$ or constant time for the kernel.

Realistically, not all threads, or streams, will be executed concurrently. Since each SM on the RTX 3090 has a limit of 41,984 concurrent threads, the amount of work to be completed for the extermination procedure is $O\left(\frac{4N^3}{41{,}984}\right)$, which assumes the maximum utilization of SMs. The formula in Eq. (6) can be used to analyze the work done by an SM. In the equation, SM_{total} is the total number of SMs on the device and $blocks_{max}$ is the total number of blocks that can run concurrently on each SM.

Before continuing, it is important to understand the effect that weight, $SM * blocks_{max} * 32$, will have on the complexity since, arbitrarily, the constant multiple would otherwise be ignored. So, it is better to give the values more context. Comparing the RTX 3090 value to $|p_{space}|$ and $|p_{grid}|$ we get: $\lfloor \sqrt[3]{41,984} \rfloor$ is 34 and $\lfloor \sqrt{41,984} \rfloor$ is 204. This provides a reference for the theoretical total number of steps required by the graphics card to complete the work.

$$work = \frac{complexity_{serial}}{SM_{total} * blocks_{max} * 32} \tag{6}$$

When compared to p_{space}, puzzles up to size 5 achieve full concurrency. While concurrency is achieved for the available work, the number of steps determined by $O\left(\frac{complexity_{serial}}{SM_{total}*blocks_{max}*32}\right)$ must split the work for puzzle sizes 6 and higher. For the kernels bounded by p_{grid}, full concurrency is achieved up to size 14. Consequently, puzzle sizes 5 and below can run concurrently throughout the entirety of the algorithm.

The thread count used in the original parallel RFS algorithm was set to 128 per block. The definition was later reduced to 64 threads per block using the dynamic thread procedure presented in the previous chapter. Dynamic implementation ensures that most puzzles will benefit from 100% utilization rate of the SMs.

The dynamic thread definition was not used to update the static definition used in the first iteration of pRFS. Therefore, to ensure full utilization rate, the base thread count should be increased to 128 per block, which would allow for 100% utilization. Due to the dynamic implementation, no puzzles were noted to have populations small enough to allocate less than 128 threads per block.

Continuing, the number of active species is then derived by the host from the remaining population prior to the execution of the next kernel. This value is copied to device memory and, the species initialization kernel is executed with the thread configuration defined by S_{total}. Upon kernel execution, each species is evaluated for existence in the S_{active} population, and the initial proportion is set accordingly.

Since we are using the $|S_{total}|$ definition for population, the number of steps required by the device to execute the kernel is bounded by $O\left(\frac{N^3}{41{,}984}\right)$. The thread complexity is $O(1)$, since each thread assesses the species label and assigns the appropriate value. The kernels used to derive average fitness and the generation update will have identical complexities.

This leads into the evaluation of the niche computation. Niche is computed by iterating over all four regions containing N elements. As there are N^3 species performing this operation, we showed a serial complexity of $O(N^4)$. This provides a thread complexity of $O(N)$, and the device requires $O\left(\frac{N^4}{41{,}984}\right)$ steps to complete the kernel.

The complexity of parallel dot product is simple to derive. First, there are S_{total} multiplications to perform. Second, parallel add reduction cuts the number of additions in half for every step. Consequently, a thread complexity of $O\left(\frac{|S_{total}|}{P} + \log_2(P) \mid |S_{total}| \leq P\right)$ is achieved.

The remaining two kernels are solution acquisition and verification. Both kernels are configured with threads defined by $|p_{grid}|$. Solution acquisition assigns each thread to a cell. The thread then searches for the best fit species associated with the cell. This is an imbalanced operation due to the variation in number of species occupying each cell.

Imbalanced data is an important point that dictates the analysis of pRFS when further augmentations are applied. Recall Figure 7 in which a problem's species distribution is shown over 3D space. There are very few species that occupy most cells in a puzzle. Similarly, there are very few cells that contain a lot of species. This extends to the number of conflicts that exist between the species. This is shown in Figure 9, which illustrates the number of conflicts across one of the size 6 puzzles.

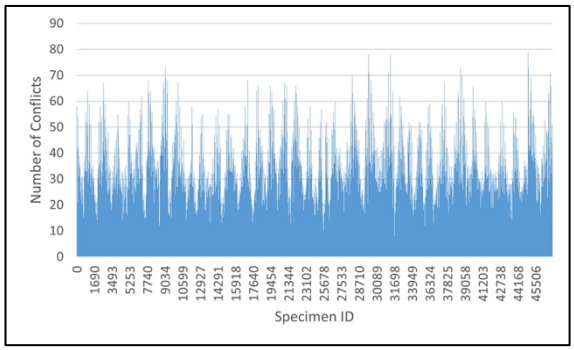


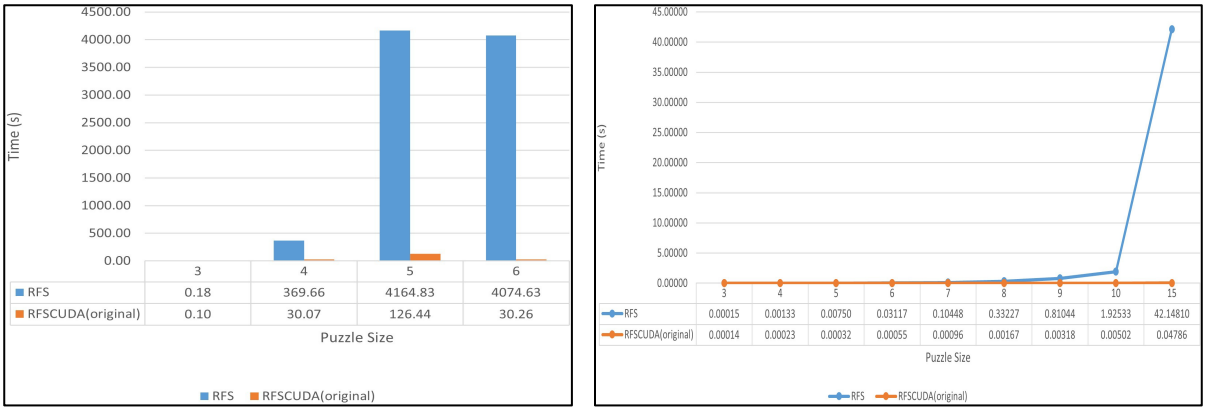
Figure 9. *Conflict Distribution per Specimen*

This makes deriving the specific complexity difficult due to the variations between puzzles. Since the full puzzle structure is retained, each thread assesses exactly N species in search of the fittest one. Effectively, we are masking the imbalanced data via the uniform computations performed on it, which provides a thread complexity of $O(N)$.

The solution verification is similar. Each thread compares the symbols of all conflicting regions to itself, setting the solved flag as false if a conflict is found. During execution, the array used to track symbol uniqueness must be reset between the evaluations of the first three regions. This gives a thread complexity of $O(N)$.

The analysis showed significant potential to reduce the complexity of serial RFS by exploiting the CUDA paradigm. The dominant factor in the analysis was the computation of the niche count, the complexity of which was $O(N^4)$. By assigning a thread to each of the species, a parallel complexity of $O(N)$ was obtained for each.

The two figures below show the overall performance enhancement achieved by the CUDA implementation. Figure 10a shows the reduction in time required to solve puzzles in the 3 to 6 size range. No puzzles beyond this range were solved with the CUDA implementation. Figure 10b gives a better assessment of overall performance by graphing the average time required to compute each generation.



a) RFS serial vs. parallel time to solve

b) RFS serial vs. parallel time per generation

Figure 10. *Performance Graphs for Serial and Parallel RFS*

Improper use of the $dim3$ structure became a major scalability issue. As a result, the analysis of the size 15 puzzle was not possible. The value reflected in the graph is derived from the average increase seen in the other iterations of the algorithm. Still, the results show significant improvements to the time of the algorithm as described in this analysis. Table 3 provides the average speed improvement over the serial implementation at each puzzle size.

Table 3. *Serial vs. Parallel RFS Speedup*

Sudoku Size	Achieved Speedup
3	1.12
4	5.83
5	23.69
6	56.78
7	109.26
8	198.51
9	255.25
10	383.36
15	880.61

8.2 Memory Optimization

Memory was optimized by reducing the population size from $O(|S_{total}|)$ to $O(|S_{active}|)$. Reducing the memory allows for more efficient use of the SMs since $O(|S_{active}|)$ closely follows the curve set by $O(|p_{grid}|)$. The graph in Figure 11 illustrates the average number of species at each puzzle size. The value of $O(|S_{active}|)$ can be derived as $O(c_{avg}|p_{grid}|)$, where c_{avg} is the average number of species per cell in p_{grid}. More specifically, $|S_{active}| = c_{avg}N^2$.

This is significant since pRFS should now be able to launch all threads concurrently for puzzle sizes below 15, using the RTX 3090 graphics card. However, a tradeoff is made since all species must now iterate over the entire population to find their conflicts. Consequently, all species must now iterate over the entire S_{active} population to find the conflicting species. As

noted at the end of the previous section, most species conflict with a relatively low number of other species. In addition, it is not uncommon for some species to have no conflicts.

One other, less pervasive change was made to the niche computation. It was noted that each species niche only included up to N species per region. Originally, this was done to limit the computational complexity of the algorithm. However, each region has up to N^2 species, which results in a large portion of the space being ignored. Yet, we know there is a relatively small number of species occupying this space. In applying the change, the maximum number of species is ensured to be utilized during the niche computation.

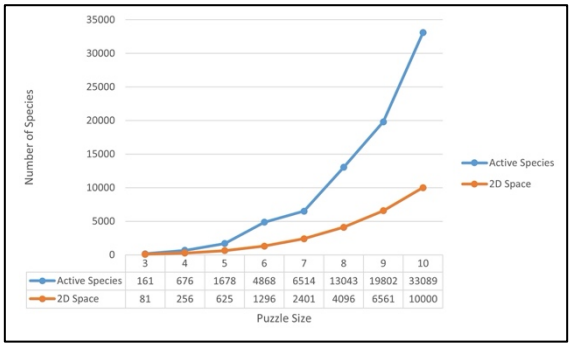


Figure 11. *Average Species Compared to 2D Space*

As in Chapter 7, we will cover the initialization procedures in the section on CDP analysis. This brings us to niche computation. There are two points to make for this version. First, proportionate selection was absorbed into the niche computation. Combining these two kernels removed the launch overhead associated with the host. Second, we can no longer use the serial complexity analysis due to how the conflict accumulation was changed.

Modifying the complexity analysis is simple; instead of iterating over N species, we iterate over N^2 species. The serial complexity then increases to $O(N^5)$. While the complexity is increased, the added work better reflects the conflict pressure on each species. However, following the memory reduction to a parallel implementation, the threads must now iterate over the S_{active} species. This changes the serial complexity again by reducing it to

$O(|S_{active}| \times |S_{active}|)$. Since we know $|S_{active}|$ is $c_{avg}|p_{grid}|$, the complexity is improved to $O(c_{avg}^2 N^4)$.

For the implementation, threads sequentially load each species into register memory. Once loaded, the species are assessed for conflict in each region. Locality is held at the forefront in this implementation as each species is moved to register memory once. By iterating over the entire population, thread complexity is set to $O(|S_{active}|)$.

The memory reduction doesn't change the algorithmic complexity of the dot product. However, to assess the current performance, the number of species computed needs to be updated, giving a complexity of $O\left(\frac{c_{avg}N^2}{P} + \log_2(P) \mid c_{avg}N^2 \leq P\right)$. This benefits the dot product greatly with a significant reduction in the number of computations required. In addition, all computations performed are useful.

The algorithmic complexity for the generation update kernel doesn't change and remains $O(1)$. The generation update does benefit from the reduction in memory as the amount of wasted work is limited to the final block executed by the kernel. This is a product of the kernel thread definition.

The solution acquisition and verification do change because of the memory reduction. Originally, the acquisition kernel iterated over the four regions, which gave a thread complexity of $O(N)$. Since the structure of the puzzle is not retained in the reduction, it falls victim to iterating over the population array. So, the thread complexity increases to $O(c_{avg}p_{grid})$, while the amount of work doesn't change. The implementation forces the device to perform $O(c_{avg}^2 p_{grid}^2)$ computations.

Solution verification received a major overhaul with the memory reduction. In the previous two implementations, each thread allocated N Boolean values to track duplicate symbols. Additionally, each region was checked independently prior to proceeding to the next.

The memory allocations are removed, and a single loop is used to check all regions. This is achieved by mapping the thread to the cell's symbol. The thread then uses the iterator to define a radius around its cell. The cells in each region that fall in this radius are checked against the thread symbol for equivalency. If a duplicate is found, the solved flag is set to false, and all threads terminate at the next iteration.

The implementation significantly reduces memory use and employs better locality and also enjoys significant redundancy as each cell is checked for duplicate symbols N times. This provides a more robust validation of the result, as each thread is only concerned about the validity of its symbol.

The application of memory optimization yields a major improvement to the memory space. Going back to Figure 8, as little as 77% to more than 99% of the memory required for tracking species was removed, which leaves only those species that are required to process the RFS algorithm.

While this presents a great result, the computation drawback was moderately significant. Most of the algorithm complexity has a lower bound dictated by the size of the active population. This presents a big issue when attempting to compute the niche count, since the entire condensed activation matrix now must be computed.

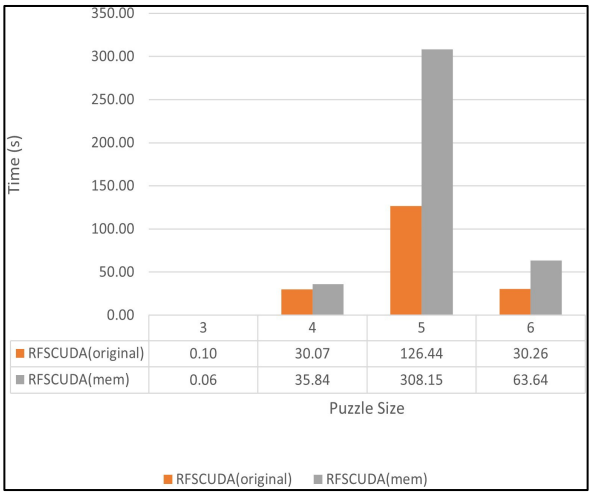
Thus, the decrease in performance presented in Figure 12a comes as no surprise. The results show the average time required to solve the puzzles in the 3 to 6 size range. Looking deeper into the performance, Figure 12b shows the average time required to compute each generation. Finally, Table 4 quantifies this decay, which shows a near five-fold decrease in performance by puzzle size 15.

While the memory optimization underperformed the previous implementation, it does achieve a moderate speedup over serial RFS. In this case, a speed factor of more than 200-fold is achieved by size 15. This shows the overall memory implementation can be considered a

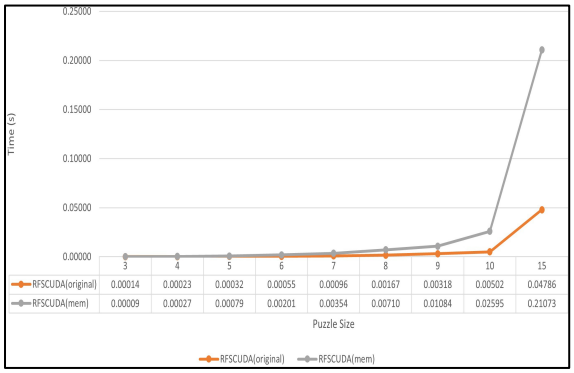
success. The algorithm effectively trades up to a fifth of its performance for an extreme reduction in the memory space used.

Table 4. *pRFS with Memory Optimization Speedup*

Sudoku Size	Speedup pRFS	Speedup Serial RFS
3	1.51	1.69
4	0.83	4.83
5	0.40	9.50
6	0.27	15.47
7	0.27	29.55
8	0.24	46.83
9	0.29	74.76
10	0.19	74.20
15	0.23	200.01



(a) pRFS vs. pRFS with memory optimization time to solve



(b) pRFS vs. pRFS with memory optimization time per generation

Figure 12. *Performance Graphs for pRFS vs. pRFS with Memory Optimization*

8.3 CDP Analysis

As the traditional CUDA model only handled static thread deployments, CDP is useful for flushing out kernels containing variable granularity. Granularity is the measure of the amount of work performed by a task. Exploiting this extended model, the exterminate and niche

computation were reconfigured due to their complexity dominance in the algorithm. The dot product and generation update kernels did not have granularity considerations, so they couldn't benefit from CDP. Solution acquisition is anecdotally evaluated, but the results are not used in the data. Finally, while solution verification has uniform work to be completed, the amount of work is typically too small compared to warp size. Thus, it wouldn't benefit from CDP.

In line with the memory optimizations, the extermination procedure was radically changed. In the original algorithm a population with S_{total} species was generated prior to running the extermination. This population is now held in a single double array with the initial values set to 0.

The host side stream allocation and deployment were removed in lieu of using CDP. The exterminate kernel uses the $|p_{grid}|$ thread definition. Each thread is subsequently mapped to a clue, and a child kernel is launched with the $|p_{space}|$ thread definition. This allows for each thread to be mapped to a symbol. If the symbol conflicts with the clue, the location in the array is set to 1. With this launch configuration, the extermination kernel achieves an $O(1)$ thread complexity. As shown in Figure 13, this provides a significant speed increase in the initialization of the algorithm.

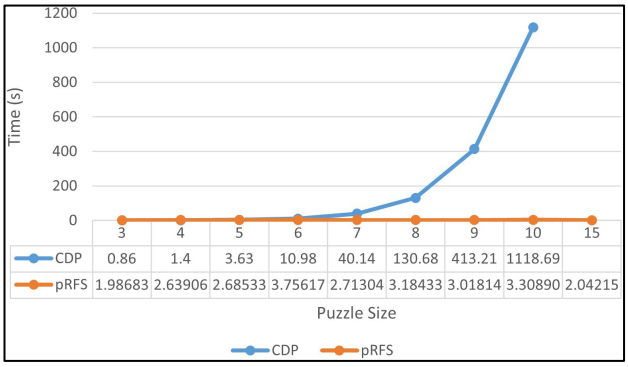


Figure 13. *CDP Initialization Time*

Once the extermination procedure completes, the host proceeds to the dot product to compute the number of species exterminated. The value is copied back to the host, which

computes the final value for $|S_{active}|$. Since the dot product was used, a thread complexity of $O\left(\frac{\mathrm{N}}{P} + \log_2(P) \mid \mathrm{N} \leq P\right)$ was achieved over the $O(N)$ serial implementation.

As most species have very few conflicts, two glaring issues emerge from applying CDP to the niche computation. Some species have little to no work, while others have significantly more—the result of which causes extreme imbalance in the work. Second, child kernels need to be uniformly launched. Otherwise, there will be divergence in the threads, resulting in catastrophic consequences. The uniform requirement may force kernel deployment whether work exists or not. If the work is there, the amount is likely very small.

The complexity analysis becomes muddied at this point since all threads technically achieve $O(1)$ performance. The atomic addition used to ensure mutual exclusion forces threads to serialize when accessing the same memory location. With this serialization, the species that has the most conflicts will dictate the complexity. Additionally, the overhead from launching all the kernels will increase the time required to complete the niche computations.

The consequence of mutual exclusion is major divergence in thread processing. Those species with little to no conflicts terminate early, while the few with significant work are forced to serialize. Subsequently, the kernel must always wait for the species with the most conflicts to finish the addition. The effects of this divergence can be seen in Figure 14.

The same rationale is used to explain why CDP won't work for solution acquisition. CDP would resolve the issue of searching for the symbols associated with each cell. However, most cells typically do not have many symbols associated with them. Thus, the atomic max, which would be used to assess fitness, forces the same trap niche in which computation fell.

Table 5 quantifies the speed factor achieved. These results, as with Figure 14, only reflect the application of CDP when used on niche computation. There is a significant reduction in computational performance across the sizes. Sizes 3, 4, and 5 present significant points of interest, with the latter two clearly seeming extraneous.

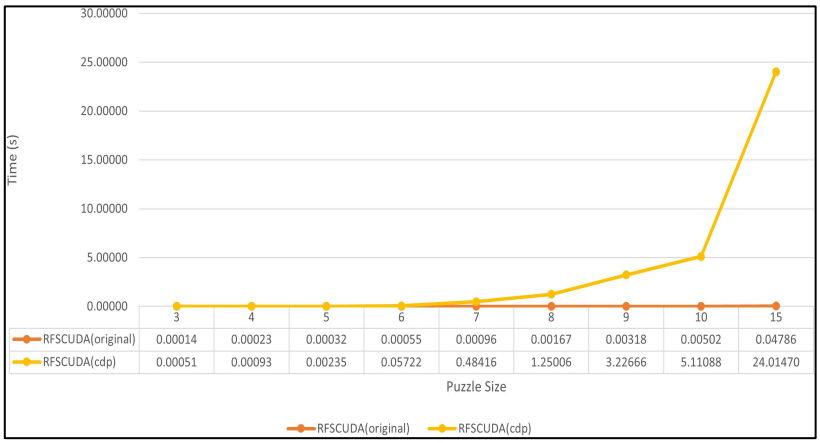


Figure 14. *pRFS vs. pRFS with CDP Time per Generation*

The most likely cause of these extraneous points is in the SM utilization. With memory reduction, most puzzles up to size 14 can spread their computations across all SMs. The number of steps the device requires to compute is subsequently held at or very close to 1. With CDP, these computations explode by many magnitudes.

The size 3 puzzles are of interest, as the explosion exasperates the already tight balance of work versus data. The first parallel implementation only achieved a 1.12-fold performance increase, which is largely due to the ampere architecture used. Assessments performed on older architectures always had a reduction in performance for this puzzle size. Using these, a drop in performance was sure to occur with the significant addition of useless work for this puzzle size.

The work to computation ratio approaches equilibrium for the size 4 puzzles, resulting in a small speed improvement. The equilibrium of work and computation seems to be achieved at size 5, providing a slightly better speed factor. This balance is shattered at the size 6 puzzles. Here, the issue transitions from SM utilization to the number of steps required by the device to perform the computations. The result is a radical drop in performance as the additional work is useless.

Table 5. *pRFS with CDP Optimization Speedup*

Sudoku Size	Speedup Serial RFS
3	0.31
4	1.43
5	3.20
6	0.54
7	0.22
8	0.27
9	0.27
10	0.25
15	0.38

8.4 Exploitation of Conflict Pairs

Applying CDP to address granularity concerns was useful for the initialization of RFS. Further application proved to degrade the algorithm's overall performance. However, it did reveal key problems in the algorithm's execution. First, the amount of work performed is very imbalanced. This results in a lot, or even most, threads doing little to no work. Additionally, a lot of time is wasted searching for conflicts. These two points culminate in a more pervasive issue—the activation matrix, hidden by niche computation, is recomputed every generation.

Going contrary to our memory optimization, let's break open the activation matrix. We can keep memory optimization at the forefront of the implementation. So far, we defined the population by only keeping the S_{active} species. This gives a condensed activation matrix size of $|S_{active}| \times |S_{active}|$ conflicts. Still, most species do not conflict with the others, so the matrix is still sparse but denser. Substituting the size $c_{avg}|p_{grid}|$ we get $c_{avg}|p_{grid}| \times c_{avg}|p_{grid}|$. Since the value of $|p_{grid}|$ is N^2, the size of the activation matrix is then shown to be $c_{avg}N^2 \times c_{avg}N^2$. Using these values, we can derive a memory bound of $O(c_{avg}^2 N^4)$, which is significantly better than the $O(N^6)$ complexity when using the p_{space} population.

The complexity can be refined further since the number of active species is known, $|S_{active}|$. While the exact number of conflicting species is currently unknown, we have been

exploiting the average number of them throughout this chapter: $c_{avg} = \frac{|S_{active}|}{|p_{grid}|}$. Practically, the average value will be an underestimate since $c_{avg} = \left\lfloor \frac{|S_{active}|}{|p_{grid}|} \right\rfloor$.

To be more precise and measure all conflicts, the matrix would have an upper bound of $O((c_{avg} + 1)^2 N^4)$. While the complexity provides a reasonable upper bound, we can just assume c_{avg} is precisely the average number of species in the $|p_{grid}|$. This is because a procedure already exists that uses this information and derives the number of conflicts each species has. It is implicitly done every generation via niche computation. So, the number of conflicts in a puzzle can be extracted from the niche procedure.

In effect, our matrix will hold the conflict key pairs for the puzzle. Each pair will track the corresponding location into the S_{active} population, which implicitly reconstructs the Sudoku structure. To further reduce memory, each pair tracks the conflict's weight, as opposed to having duplicate key pairs. Along this same vein, species know their proportion and weight, since they fully conflict with themselves. This immediately removes the need to keep $4|S_{active}|$ key pairs in memory.

The execution of this implementation starts with two kernels added to the initialization procedure. The first kernel extracts the niche computation out of the generation loop. Instead of modifying species proportion, it counts the number of conflicts for each species. Consequently, the niche search complexity is added to the initialization and only occurs once.

The second kernel derives the parallel add reduction by Mark Harris (Harris, M., 2008) and is derivative because the kernel is templated to perform three operations: add, max, and min. The add reduction is an implementation of the structure he presented. The kernel was templated to accommodate max and min reduction for the acquisition of conflict extremes in the population. This assigns complexity of $O\left(\frac{k}{P} + \log_2(P) \mid k \leq P\right)$ to each thread for all three operations. In this case, k is the number of conflicts in the population.

Now the pairwise conflict matrix is allocated, and a kernel launched with thread configuration defined by the number of conflicts. Each species starting conflict ID is computed by summing the number of conflicts of the previous species in the population, which then iterates over the population, tracking both conflict status and weight of the conflict. This ensures minimum memory is required as described above. In total, each thread yields complexity $O(2|S_{active}|)$.

With all conflicts held in memory, the niche kernel can now be modified to accumulate the niche count for all species in the active population. The first implementation uses a thread configuration defined by the total number of conflicts, c_{total}. A second thread configuration is defined by S_{active}, which is used in the emulation of RR load balancing.

In the first kernel, all threads are gated, so exactly c_{total} are mapped to all conflicts. The conflict pairs are pulled into register memory where the partial niche is computed. The niche is then added back to the total niche for the primary species. The addition is performed atomically on global memory, which masks the data transfer of the partial niche.

This presents a thread complexity of $O(1)$. Again, the complexity is deceptive since the atomic additions are serialized on global memory and, thus, imbalanced. However, the issues presented in CDP are resolved as all threads perform useful work. Furthermore, the work is better distributed over all SMs, providing a thread complexity of $O(c_{avg})$.

The load balancing kernel mostly executes the operations as previously described. The thread definition is changed to map all threads required to compute and assign the partial niches. A loop is added to ensure threads do not exceed the total number of conflicts, which provides three primary benefits. First, all threads defined are utilized. Second, all threads are maximally utilized across the available work. Finally, the loop provides a natural guard on the memory space. The time to solve shown in Figure 15 reflects the reduction time acquired from RR load balancing.

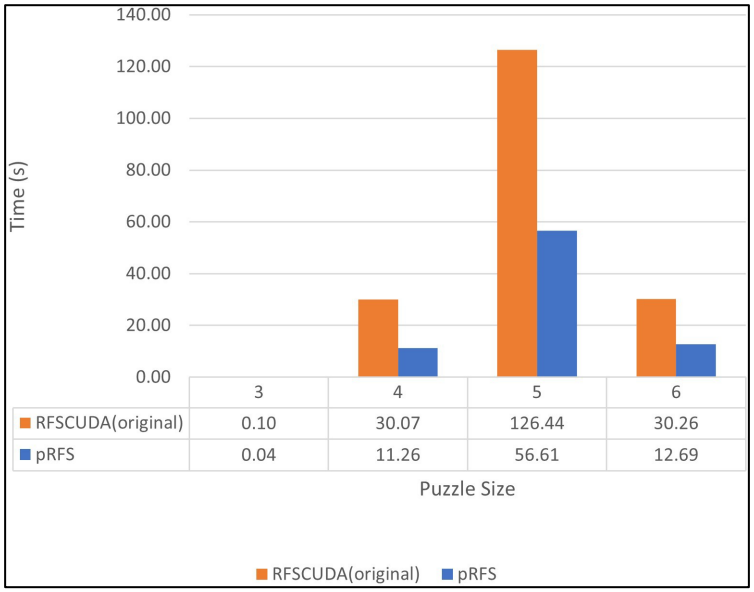


Figure 15. *pRFS vs. pRFS with RR LB Time to Solve*

Atomic add remains a major drawback on the total efficiency for both kernels. The non-load balancing kernel mitigates the issue by distributing the serializations over the available SMs. Therefore, some normalization is achieved across the device due to hardware-imposed scheduling.

Load balancing provides better control of this normalization process in two ways. First, the $|S_{active}|$ thread definition is used to provide sufficient parallelization of the problem's complexity space. In addition, all threads deployed can be utilized without gating.

Second, non-load balancing relies on the device's scheduling mechanism to disrupt the locality of the serial computations—load balancing takes some of the control away from it through the looping mechanism. Each iteration of the loop distributes the computations across all SMs, providing better normalization of the parallel and serial computations. In doing so, the amount of time required for each SM to wait for serial computations to complete is minimized.

The figures below demonstrate the various metrics used for quantifying the performance of the conflict key pairs and round robin load balancing. Figure 16 expands the solution metrics by presenting the time to compute each generation.

Table 6 provides the speed factors achieved over the original parallel and the serial RFS algorithms. These figures are a combination of using conflict pairs to reconstitute the problem space and round robin for better balancing of the computations across the SMs. We see an average speed factor of 2.81 against the parallel implementation.

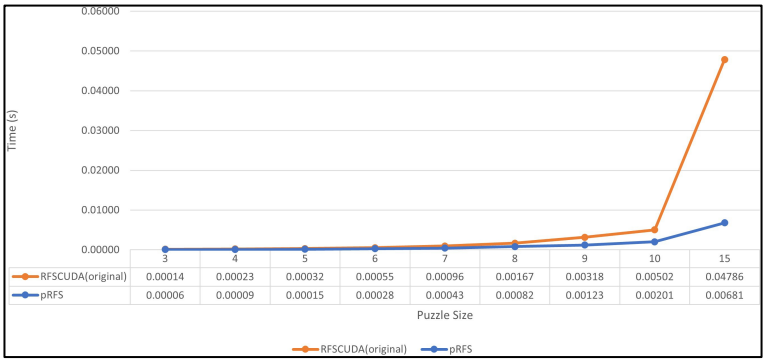


Figure 16. *pRFS vs. pRFS with RR LB Time per Generation*

Table 6. *RR Load Balancing Speedup over Original pRFS and over Serial RFS*

Sudoku Size	Speedup pRFS	Speedup over serial RFS
3	2.14	2.40
4	2.62	15.26
5	2.17	51.36
6	1.97	112.03
7	2.20	240.51
8	2.03	942.03
9	2.59	660.95
10	2.50	958.37
15	7.03	6186.53

The graph does not address whether load balancing had any effect, though. As indicated by Table 7, puzzle sizes 3 to 5 had a small decrease in performance over the non-load balancing implementation. With the relatively small computation space presented by these sizes, the SMs

likely processed the computations required by the kernel in one cycle. Therefore, attempting to load balance the computations result in a reduction in SM utilization, degrading performance slightly.

On the other hand, the computation space for sizes 6 and above achieve enough utilization to make load balancing worthwhile. As the puzzle sizes increase, load balancing becomes more beneficial, with the size 15 puzzles receiving a 23.5% increase in performance over non-load balancing.

While this shows promising results for the application of load balancing, more research is required to better balance the computational and memory requirements. Between the two approaches of memory optimization and greedy memory, there should exist a balanced parallel RFS algorithm that optimally addresses memory and computation—a target implementation of which is addressed in Section 9.1.2.

Table 7. *pRFS Time with No Load Balancing and RR Load Balancing*

	No Load Balancing		RR Load Balancing		
Puzzle Size	**Avg Elapsed time**	**Gen Time**	**Avg Elapsed time**	**Gen Time**	**Speedup**
3	60.1143	0.000062	63.4904	0.000065	0.961742
4	83.3029	0.000082	85.5320	0.000087	0.945898
5	139.0670	0.000140	146.0100	0.000146	0.960105
6	353.2200	0.000300	324.3390	0.000278	1.077780
7	484.7760	0.000485	434.4050	0.000434	1.115954
8	974.8520	0.000975	822.5690	0.000823	1.185131
9	1401.2700	0.001401	1226.1700	0.001226	1.142802
10	2315.7900	0.002316	2008.9700	0.002009	1.152725
15	8415.3200	0.008415	6812.8800	0.006813	1.235207

8.5 Computational Noise

Better management of conflict noise was attempted by modifying the initial proportion. This experiment was to assess how varying the initial proportion affected overall performance. While it would have been ideal, none of the settings affected the solution rate of the algorithm.

Those values used in this experiment were ensured to remain between 0 and 1 as prescribed in the literature. Table 8 details the baseline times achieved using the original uniform distribution. For the evaluation of this experiment, an error rate of $\pm 1\%$ is used to accommodate timing fluctuations due to device performance. The values for "Avg. Gen" are the average generations required to solve.

Table 8. *Performance of Uniform Distribution for Initial Proportion*

Sudoku Size	Avg. Gen	Time to Solve (s)	Avg. Gen time (s)
3	675	0.0445	0.000066
4	127414	11.2645	0.000088
5	387782	56.6051	0.000146
6	54683	12.6893	0.000232

Table 9 gives results where the species were uniformly assigned a value of 1. This assignment provided a moderate reduction in the average generations required to find a solution at the 3 and 6 puzzle sizes. These puzzles had an average generation reduction of 11.26% and 13.45%, respectively. A very minor reduction of 1.12% and 0.64% was seen at sizes 4 and 5.

Table 9. *Performance of 1 for Initial Proportion*

Sudoku Size	Avg.Gen	Time to Solve (s)	Gen time	ΔAvg.Gen	ΔTime to Solve (s)	ΔGen time
3	599	0.0393	0.0000657	88.741%	88.455%	99.678%
4	125991	11.0810	0.0000880	98.883%	98.371%	99.482%
5	385319	56.0205	0.0001454	99.365%	98.967%	99.600%
6	47326	10.8873	0.0002300	86.546%	85.799%	99.137%

The average solve time saw equivalent figures to average generation with variance between 0.28% to 0.75% and a median value of 0.486%. The time per generation was equivalent to uniform distribution with a variance of 0.32% to 0.87%. A median time of 0.526% was achieved. The variations exhibited between puzzle sizes is likely due to the extremely small sample space used for testing the larger puzzles. As the size 3 puzzles involved the most

robust sample space, similar results should be achieved by increasing the samples for each of the larger sizes.

Table 10 represents the case for which the initial proportion is assigned $\frac{1}{s_c}$. This setting provided a variance range of -1.21% to 0.15% for the average generations. With such a small range, the value likely emulates the uniform distribution with a minor degradation. Expanding the sample space would be necessary to validate this correlation. Comparably, the average generation time for sizes 3 to 5 varied between -0.62% to 0.76%. The size 6 puzzle introduced a divergence to these figures with a 2.023% reduction in time to solve. Including this value, a median change of 0.695% is noted.

Table 10. *Performance of* $\frac{1}{s_c}$ *for Initial Proportion*

Sudoku Size	Avg. Gen	Time to Solve (s)	Gen time	ΔAvg. Gen	ΔTime to Solve (s)	ΔGen time
3	674	0.0441	0.000065	99.852%	99.243%	99.390%
4	127504	11.1957	0.000088	100.071%	99.389%	99.319%
5	392454	56.9508	0.000145	101.205%	100.611%	99.413%
6	54713	12.4326	0.000227	100.055%	97.977%	97.923%

The time per generation saw a slight increase with a variation between 0.61% to 2.08%. The divergence in data occurs again at size 6 with a value of 2.077%. An achieved median value of 0.989% was seen, indicating roughly equivalent metrics to the uniform distribution. While the metrics appear to closely follow those achieved via the uniform distribution, a larger sample space is required to empirically validate this observation.

Table 11 addresses the opposing probability to Table 10 with an initial proportion assignment of $1 - \frac{1}{s_c}$. The metrics associated with this value were surprising due to the better performance. The average generations saw a reduction variance between 0.63% to 13.5% with a median of 7.824%. The size 5 puzzles saw the least reduction at 0.639%, and the size 6 puzzles had a 13.496% reduction.

Table 11. *Performance of* $1 - \frac{1}{s_c}$ *for Initial Proportion*

Sudoku Size	Avg. Gen	Time to Solve (s)	Gen time	ΔAvg. Gen	ΔTime to Solve (s)	ΔGen time
3	597	0.0392	0.000066	88.444%	88.256%	99.787%
4	120272	10.5696	0.000088	94.395%	93.831%	99.403%
5	385305	55.9970	0.000145	99.361%	98.926%	99.562%
6	47303	10.9417	0.000231	86.504%	86.228%	99.680%

The average time to solve yielded a slightly better median of 8.19% with a variance between 1.07% to 13.78%. Puzzle sizes 5 and 6 remained at the extremes of this range. The average generation time for this setting was the tightest with a median value of 0.392%. This indicates the setting provides better overall performance, and the improvement appears to be stable along puzzle sizes.

The final setting, $\frac{(|p_{space}|\%s_c)+1}{s_c}$, is reflected in Table 12. This setting had the largest variation for average generation and average solution time. The average generations required to solve ranged from -46.89% to 26.4%. Puzzle sizes 3 and 6 had significant reduction in generations required to solve at 28.148% and 26.390% respectively. Puzzle sizes 4 and 5 saw an increase in generations required at -3.613% and -46.884%. While the range is very large compared to the other settings, the median reduction is 1.01%.

Table 12. *Performance of* $\frac{(|p_{space}|\%s_c)+1}{s_c}$ *for Initial Proportion*

Sudoku Size	Avg. Gen	Time to Solve (s)	Gen time	ΔAvg. Gen	ΔTime to Solve (s)	ΔGen time
3	485	0.0319	0.000066	71.852%	71.704%	99.794%
4	132017	11.6524	0.000088	103.613%	103.443%	99.836%
5	569591	82.9220	0.000146	146.884%	146.492%	99.733%
6	40252	9.2201	0.000229	73.610%	72.660%	98.710%

The average time to find a solution understandably follows this range at -46.49% to 28.3%. A median reduction of 1.425% is noted. All puzzle sizes align with the values reported

for the average generations to solve. As with the other settings, the average time per generation did not diverge much from the uniform distribution. A reduction range of 0.2% to 1.3% occurred with a median change of 0.482%.

These settings make it clear that the uniform distribution is not the ideal setting. Both 1 and $1-\frac{1}{s_c}$ provided the most clear-cut improvement with the latter achieving the best performance overall. However, the $\frac{(|p_{space}|\%s_c)+1}{s_c}$ setting provided the most significant reduction and increase. The setting $\frac{1}{s_c}$ had the more surprising result, as it underperformed the counter probability.

Why did this setting underperform? One thought is that the counter probability reflects the counter pressure on the species. This pressure is very minimal on the species as the niche computation is applied. So, introducing the pressure as a starting point better reflects the overall pressure experienced by the species. Consequently, utilizing $\frac{1}{s_c}$ would appear to overfit the conflict pressure of each species as the niche computation is applied.

While adjusting the initial proportion did not yield an increase in the number of puzzles solved, it showed significant potential in reducing the number of generations. Consequently, more research should be conducted to better fit the initial proportion. Research of this kind should aid in the overall reduction in time required for RFS to find a solution.

CHAPTER 9

DISCUSSION

With the comprehensive analysis of RFS completed, and by extension niching algorithms, some extraneous topics remain. To address these topics, this chapter expands on ideas and concepts that are viable for further research for (p)RFS. Section 9.1 addresses the subtle concept of uniqueness, which, while pervasive, was heavily ignored. Techniques that may improve solution acquisition are discussed in Section 9.2. Finally, Section 9.3 updates the algorithm in Figure 4 to aid in the deployment of an efficient parallel RFS algorithm.

9.1 Considering the Effects of Uniqueness

This section investigates the quality of uniqueness as used in the context of both Sudoku and RFS. Two primary points are addressed during this investigation. In the first section, the argument for including symbol uniqueness as a constraint is presented. The following section presents a method for reducing the niche computation. In the conclusion of the section, a serial and parallel complexity analysis is derived for the proposed change. The concepts addressed throughout the chapter are illustrated through mental exercises to better formulate and clarify the points.

9.1.1 Symbol Uniqueness: A Fifth Constraint

Uniqueness was a major, albeit subtle, aspect of Sudoku that had massive consequences for the application of (p)RFS. However, given the significance in its role, the concept is largely ignored for most of the algorithm. Only two key areas explicitly introduce any use of the concept—species extermination and solution validation. The remaining algorithm focuses on the species relation to the constraints and their respective fitnesses.

Since the cell is one of the constraints, it can be the linchpin to arguing that RFS does consider uniqueness. But what uniqueness does this constraint encapsulate? Is the cell unique?

Yes. Is the symbol in the cell unique? Also, yes. The more important question then is: Does RFS treat the cell as unique, or the symbol?

The evaluation of noise triggered a rather intriguing question. Why is symbol uniqueness not treated as one of the constraints? Granted, casting this rule as a constraint is not as clear as performing partitions on 3D space, which the four rules inherently do. However, how to handle symbols, in terms of uniqueness, is clearly stated in the Sudoku rules.

Consequently, adding a fifth constraint would allow for the partitioning of species based on symbol uniqueness. How would this new constraint be enforced? Would the accumulation for niche computation be on the partitioning of sameness? This seems counterintuitive as we're looking for uniqueness, which is distributed along the remaining four regions.

Contrarily, would this constraint accumulate all symbols that are different, treating the uniqueness of the symbol in question as absolute? This seems plausible since it may better model the statistical distribution of the species conflict pressure.

Maybe this assessment is an over-analyzation of the problem's rule space. Since the evaluation of uniqueness can inherently be masked by the previous four constraints, it may lead to a potentially fruitless endeavor. The argument against this brings the dilemma full circle.

The four constraints may mask uniqueness. RFS wouldn't find solutions if it didn't. But what uniqueness is being masked? Each of the four regions intersect at a unique cell. Is it the symbol of this cell that is being treated as unique or the cell? So, can uniqueness be a constraint that was overlooked?

9.1.2 Niche Redefined: Identity and Importance of the Cell

Uniqueness seems to be particularly powerful in the domain of Sudoku, which is understandable since it was built into the rules. Still, it is not always obvious how a simple concept can have such pervasive consequences. The last section presented arguments for and against uniqueness as a constraint. How else is uniqueness relevant then? Regarding Sudoku,

nothing else really comes to mind. For RFS, though, there are several constructs enforced that have uniqueness—identity.

RFS enforces niching, which enforces speciation and, in turn, identifies specific, key resources. Each species has an identity and location in 3D space. This location is some distance away from a Sudoku grid cell and represents a symbol. RFS is attempting to identify if this symbol belongs to this cell. So, RFS has been extrapolating uniqueness and making associations to search for a solution to the puzzle.

The cell is obviously the centrifuge that dictates this process. We want to know what symbol must go there. RFS achieves the answer by performing niche computations and weighing the fitnesses of species. A species fitness already has identity, though, so how can this concept be exploited by the niche computation?

Niche computation performs most of its work in searching for species. But why? Why does RFS need to search to compute the niche of each species? Well, it doesn't. The greedy memory approach gave a method to remove all but one search for conflicts. Why can't we just remove the remaining search?

To achieve this, RFS would need to focus on the identity of the cell. This cell has an association to some number of species. Those species have an identity defined by the symbol they represent. So, have those species accumulate their proportions into this cell. Now there is a partial niche computation associated with the cell. As this is possible for all cells, the entire Sudoku grid contains N^2 partial niche computations.

Since every cell contains the total proportion of all its species, RFS needs to only accumulate the partial sums across each of the regions. Algorithmically, this presents a complexity of $O(N^3 + 4N + 4)$. Why? Because there are N^2 cells in the grid. Each cell accumulates at most N species proportions. Then, each region accumulates the N partial sums

to obtain its total sum. Finally, each species computes its niche count by accumulating the region sums associated with its location.

Now, with pRFS, the process is the same, but we have enough threads to make these computations in parallel. The entire process gets reduced to two parallel add reductions and four constant additions per thread. Since pRFS has at least $|S_{active}|$ threads, each thread is mapped to a unique species. All partial sums of the cells can be done with parallel add reduction.

At this point, whether there are 0 or N species associated with a cell is a moot point. Consider that the largest puzzle size is 15, which yields $N = 225$. Take the log of 225, $N = \log_2(225) = 7.81$. The partial sums for the cells will be completed in at most eight steps for the available puzzle sizes. Similarly, each region has exactly N partial sums to complete. Again, use parallel add reduction and the same result is achieved.

Consequently, the niche computation for RFS has been reduced to $O\left(2\left(\frac{N}{P} + \log_2(P)\right) + 4|N \leq P\right)$ per thread. Going back to the max puzzle size being 15, thread computations for the available puzzle sizes has been reduced to $O(12)$ or near constant time.

In addition to the complexity improvement, memory is minimally affected. The S_{active} population is retained, which maintains the spatial benefits obtained in the memory optimization of RFS. There is potential for a slight increase in memory, share memory specifically.

Instead of performing all computations on global memory, allocate $4N$ doubles using dynamic shared memory on the kernel launch. Since there is 48KB of shared memory per SM, dynamic memory can be exploited in this manner up to $N = 1500$, or size 38 puzzles. To process any higher, one region would have to be computed at a time. This provides an upper limit of $N = 6000$ or size 77 puzzles.

Applying shared memory in this way will require some additional computations. This is due to each SM having a partial sum for each of the regions. The additional computations are limited to the number of SMs. Still, the performance benefits of using dynamic shared memory would significantly outweigh the additional computations.

9.2 Improving Solution Rate for RFS

Serial RFS was shown to have an extremely high processing time, reaching up to 56 seconds to perform one generation on a size 15 puzzle. Since the algorithm's processing time was so high, it became difficult, or even impractical, to attempt larger puzzle sizes. To resolve this, the research up until now has primarily focused on the parallel, spatial, and temporal optimizations of the algorithm. Consequently, improving efficacy has been a secondary goal while a more efficient algorithm was developed.

To address RFS's efficacy, this chapter details a couple of ideas that may improve the algorithm's ability to find a solution. Section 9.2.1 considers the implementation of gradient ascent, which is used to derive supersets of the original puzzle. Employing this technique would resolve noise induced by the combinatoric explosion of the solution space. Section 9.2.2 looks at the duplicate symbols derived from the solution acquisition and then employs the same survival-of-the-fittest technique to perform breadth-first search of the k best fit species for each cell.

9.2.1 Gradient Ascent

One method that may aid in addressing the efficacy of RFS is gradient ascent. This idea stems from the noise reduction addressed in chapter 7.4. As noise remains an issue, the effects it has on Sudoku becomes more pervasive as puzzle size increases. This is exhibited in the combinatoric explosion of the solution space. Can the effects of this explosion be limited then?

It should come as no surprise that there are various tricks that can be exploited to make solving Sudoku easier. One method is to list the available symbols in each cell, just as RFS does. RFS lacks ingenuity though, as it's not a person. A common occurrence of Sudoku puzzles is there may be one or more trivial answers or symbols that have no conflicts.

Consequently, the symbol can be fixed to the cell, which removes any occurrence of the same symbol in the regions. In effect, a superset of the original puzzle is acquired for very little work. This brings up an additional point: Why not exploit this technique as a pre-processing of the data? If the puzzle has such a trivial solution, then it's ill-suited to RFS.

To employ this technique, a balanced approach would be needed. First, a tentative solution needs to be extracted from the species, which already happens. Next, some metric needs to be used to derive a confidence level in the selected species. The metric will provide a basis for deciding if the species symbol should be fixed.

With symbols having a known trivial case, two separate metrics can be used. The trivial case is when a species has no conflicts. Extending this practice, once a species is fixed, the solution needs to be reevaluated after the extermination of the fixed species. Beyond potentially outright solving a puzzle, the trivial check minimizes the number of generations required by RFS.

Now that a base case has been set, another metric needs to be derived to reflect the confidence that a species is part of the solution. As the survival of the fittest is already used for solution acquisition, why not use the same thought process here? With the solution extracted, the only concern is whether there are duplicate symbols between intersecting regions.

So, use survival of the fittest to evaluate all species whose symbol conflicts within the four intersecting regions, which provides the most likely candidate for the solution. However, a confidence threshold is required to provide a high level of surety that the right symbol is

selected. Without this surety, RFS will have a high risk of going down a solution branch that will be wrong.

Further research will be needed to flush out what metric to use. However, one idea would be to assess if there is a significant divergence in proportion between the top two species. Consider whether the best fit species has twice the proportion of the second-best species. This would provide a reasonable confidence to set the best fit species and exterminate the rest.

What benefits would gradient ascent provide then? The most obvious is a minimization of the required generations needed for species to converge on a solution. A similar benefit would come from fixing the symbols. Each time a symbol is fixed, a superset of the original puzzle is acquired, which reduces the solution space. So, exploit this reduced solution space by implementing gradient ascent after the solution acquisition and verification. Doing so would provide RFS time to find a solution, while gradient ascent directs RFS down the most probable path to find a solution.

9.2.2 K Best Fit

Survival-of-the-fittest method has been the go-to method for extracting a potential solution from the species population. This idea can be modified slightly to improve the extractions' efficiency. Currently, the procedure arbitrarily extracts the best fit species for each cell. Once found, the solution is checked for viability.

While performing the solution acquisition in this way has proven the capabilities of RFS, its simplicity is too restrictive. The procedure completely ignores when duplicate symbols are extracted from the population. To resolve this, once the initial solution has been extracted, check to see if there are duplicate symbols in the conflicting regions. Apply survival-of-the-fittest a second time to find which of these duplicate symbols has the highest proportion. Use that symbol in the solution. For the remaining symbols, flip their proportions with the second-best fit species for their cell.

This can be extended to a k best fit model. However, the general assumption in making this modification is that RFS has generally succeeded in partitioning the solution space to find a solution.

The reason a solution isn't being found is because of the way it is extracted. So, while a k best fit model may be used, the solution is likely to exist within the two to three best fit species of each cell. Subsequently, searching any of the weaker species is likely a fruitless endeavor.

9.3 Efficient Parallel RFS Template

Below is an update to the RFS pseudo code for implementing the changes proposed in this thesis. In Section 9.3.2, bullets are added to include the solution acquisition and verification. Solution acquisition is detailed using the best fit species approach. While all sections include the parallel procedure, Section 2.a, niche computation, reflects the proposed change discussed in Section 9.1.2.

9.3.1 EPRFS Algorithm

1. INITIALIZE:
   a. Generate initial set of species (unique chromosomes) S;
   b. $\forall x, y \in S$: calculate pairwise intersection and store as $f_{x,y} \coloneqq \frac{|x \cap y|}{|x|}$
      i. launch $|S|$ threads to do $f_{x,y} \coloneqq \frac{|x \cap y|}{|x|}$ per thread.
   c. $\forall s \in S: p_s \coloneqq \frac{1}{|S|}$; // Uniform distribution across all species, initially.
      i. launch $|S|$ threads to perform parallel add reduction on S.
      ii. launch $|S|$ threads to assign $p_s \coloneqq \frac{1}{|S|}$.
2. LOOP: while (termination condition is false) do
   a. $\forall x \in S$; Evaluate and store shared fitnesses as $f_{sh}(x) \coloneqq \left(\sum_{\forall y \in S} p_y * f_{x,y}\right)^{-1}$;
      i. launch $|S|$ threads to perform parallel add reduction on $p_y * f_{x,y}$
   b. Calculate and store average shared fitness as $\overline{f_{sh}} \coloneqq \sum_{\forall x \in S}\left(p_x * f_{sh}(x)\right)$;
      i. launch $|S|$ threads to perform parallel add reduction on $p_x * f_{sh}(x)$
   c. Calculate next generation species proportions p' as $\forall x \in S: p'_x \coloneqq p_x * \frac{f_{sh}(x)}{\overline{f_{sh}}}$;
      i. launch $|S|$ threads to assign $p'_x \coloneqq p_x * \frac{f_{sh}(x)}{\overline{f_{sh}}}$
   d. Move to next generation by updating species proportions as $\forall x \in S: p_x \coloneqq p'_x$
      i. launch $|S|$ threads to assign $p_x \coloneqq p'_x$

Figure 17. *EPRFS Algorithm*

9.3.2 EPRFS Algorithm for Sudoku

3. INITIALIZE:
   a. Generate initial set of species (unique chromosomes) S;
   b. $\forall c \in p_{clues}, \forall x, y \in S$: calculate pairwise intersection and store as $f(c) = f_{x,y} \coloneqq \frac{|x \cap y|}{|x|}$
      i. launch $|p_{clues}|$ threads to launch $|S|$ threads to perform $f_{x,y} \coloneqq \frac{|x \cap y|}{|x|}$ per thread.
   c. $\forall s \in S: p_s \coloneqq \frac{1}{|S|}$; // Uniform distribution across all species, initially.
      i. launch $|S|$ threads to perform parallel add reduction on S.
      ii. launch $|S|$ threads to assign $p_s \coloneqq \frac{1}{|S|}$.
4. LOOP: while (termination condition is false) do
   a. $\forall x \in S, \forall c \in p_{grid}, \forall x \in c$; Evaluate and store shared fitnesses as $f_{sh}(x) \coloneqq \left(\sum_{\forall y \in S} p_y * f_{x,y}\right)^{-1}$;
      i. launch $|S|$ threads to perform parallel add reduction on $\forall x \in c$
      ii. with p_{grid} threads perform parallel add reduction $\forall regions \in p_{grid}$
   b. Calculate and store average shared fitness as $\overline{f_{sh}} \coloneqq \sum_{\forall x \in S}\left(p_x * f_{sh}(x)\right)$;
      i. launch $|S|$ threads to perform parallel add reduction on $p_x * f_{sh}(x)$
   c. Calculate next generation species proportions p' as $\forall x \in S: p'_x \coloneqq p_x * \frac{f_{sh}(x)}{\overline{f_{sh}}}$;
      i. launch $|S|$ threads to assign $p'_x \coloneqq p_x * \frac{f_{sh}(x)}{\overline{f_{sh}}}$
   d. Move to next generation by updating species proportions as $\forall x \in S: p_x \coloneqq p'_x$
      i. launch $|S|$ threads to assign $p_x \coloneqq p'_x$
5. CHECK SOLUTION
   a. $\forall c\ \varepsilon\ pgrid, \forall s\ \varepsilon\ S, \forall s\ \varepsilon\ c$: perform parallel max reduction
      i. launch $|p_{grid}|$ threads to perform parallel add reduction $\forall s\ \varepsilon\ c$
   b. $\forall [x, y, z]\ \exists !\ [z]\ \varepsilon\ [x, y]\ \varepsilon\ p_{grid}$
      i. launch $|p_{grid}|$ threads to assess $![z]\ \varepsilon\ [x, y]$

Figure 18. *EPRFS Algorithm for Sudoku*

9.3.3 EPRFS Algorithm Complexity

6. INITIALIZE:
   a. Generate initial set of species (unique chromosomes) S;
   b. $O(clues \times |S_{total}|)$ launch |clues| threads: $O(|S_{total}|)$ launch $|S_{total}|$ threads: $O(1)$
   c. $O(2|S_{total}|)$ launch $|S_{total}|$ threads: $O\left(\frac{|S_{total}|}{P} + \log_2(P)\ ||S_{total}| \leq P\right)$
7. LOOP: while (termination condition is false) do
   a. $O(p_{grid} \times N)$ launch $|S_{active}|$ threads: $O\left(\frac{N}{P} + \log_2(P)\ |N \leq P\right)$
   b. $O(|S_{active}|)$ launch $|S_{active}|$ threads: $O\left(\frac{|S_{active}|}{P} + \log_2(P)\ ||S_{active}| \leq P\right)$
   c. $O(|S_{active}|)$ launch $|S_{active}|$ threads: $O(1)$
   d. $O(|S_{active}|)$ launch $|S_{active}|$ threads: $O(1)$

Figure 19. *EPRFS Algorithm Complexity*

9.3.4 EPRFS Algorithm Complexity for Sudoku

8. INITIALIZE:
   a. Generate initial set of species (unique chromosomes) S;
   b. $O(\text{clues} \times |S_{\text{total}}|)$ launch |clues| threads: $O(N^3)$ launch N^3 threads: $O(1)$
   c. $O(2N^3)$ launch N^3 threads: $O\left(\frac{N^3}{P} + \log_2(P) \,|\, N^3 \leq P\right)$
9. LOOP: while (termination condition is false) do
   a. $O(N^2 \times (3N^2 + N))$ launch $|S_{\text{active}}|$ threads: $O\left(\frac{N}{P} + \log_2(P) \,|\, N \leq P\right)$
   b. $O(|S_{\text{active}}|)$ launch $|S_{\text{active}}|$ threads: $O\left(\frac{|S_{\text{active}}|}{P} + \log_2(P) \,||S_{\text{active}}| \leq P\right)$
   c. $O(|S_{\text{active}}|)$ launch $|S_{\text{active}}|$ threads: $O(1)$
   d. $O(|S_{\text{active}}|)$ launch $|S_{\text{active}}|$ threads: $O(1)$
10. EVALUATE SOLUTION
   a. $O(N^2 \times N)$ launch N^2 threads: $O\left(\frac{N}{P} + \log_2(P) \,|\, N \leq P\right)$
   b. $O(N^2 \times N)$ launch N^2 threads: $O(N)$

Figure 20. *EPRFS Algorithm Complexity for Sudoku*

CHAPTER 10

CONCLUSION

This thesis presented several modifications to the RFS algorithm to assess and optimize the spatial and temporal complexity of the algorithm. A pRFS algorithm was described and baseline metrics acquired to compare how each modification changed the behavior of the algorithm. Additionally, attempts at better modeling the conflict pressure was used to address computational noise when co-evolving species. Finally, load balancing was used to A) assess amenability of NGAs for the exploitation of load balancing techniques and B) further reduce the time required to solve by increasing SM efficiency. Load balancing provided up to 23.5% speed improvement when only assessed on the niche computation, showing it can effectively be used to increase the performance of NGAs.

While most modifications presented concrete benefits, CDP was an outlier that showed no attainable benefit. However, we established that RFS conforms to the problem domain, so it is more likely CDP is not amenable to Sudoku. For those problems that can exploit RFS as well as CDP, the results may reflect differently. Further research should be performed to validate this conclusion.

A sizable reduction in speed was noted when memory optimization was performed, but a speed improvement over serial RFS was still maintained. Assessing the opposing end of this spectrum, the potential speed increase available to pRFS was very significant, with a fourfold increase over the serial implementation.

Additionally, spatial complexity assessment in terms of minimization and maximization proved fruitful in deriving an (e)PRFS algorithm. The proposed change in Section 9.1.2 reduces the complexity of the niche computation from $O(N^5)$ to $O\left(N^2 \times (3N^2 + N)\right)$ for the serial implementation. Consequently, ePRFS obtains a thread

complexity of $O\left(\frac{N}{P} + \log_2(P) \,|N \leq P\right)$, which is in line with established parallel reduction complexity.

The solution acquisition and validation procedures are only required once, so the algorithm's performance is not reliant on them. If we then ignore them in kind, the algorithm sustains total time complexity of $O\left(\frac{|S_{active}|}{P} + \log_2(P) \,||S_{active}| \leq P\right)$ for (e)PRFS and $O(N^3)$ for serial RFS.

With the focus on the time complexity of the procedures, the looping mechanism has been ignored. While the number of generations used was an arbitrarily large number, the convergence time has been shown to be logarithmic (Horn & Goldberg, 1998). This was shown for two species, and further research is needed abstract this result to any number of species. Consequently, applying the convergence complexity gives the lower bound for (e)PRFS as presented in Eq. (7).

$$O\left(\left(\frac{(S^2-1)}{r_f S - S^2 + S - 1} ln\left(\frac{\left(\frac{1}{N-1} - r_f + S + \frac{r_f S}{1-N}\right)\left(N r_f S + r_f S + S - r_f - N - 1\right)}{\left(r_f + 1\right)^2 (S-1)^2}\right) + \frac{S}{S - r_f} ln\left(\frac{N - N r_f S - r_f S - S + r_f + 1}{(N-1)(r_f S - NS + S + N r_f - r_f - 1)}\right)\right)\left(\frac{|S_{active}|}{P} + log_2(P)\right)\right) \quad (7)$$

Finally, noise consideration proved beneficial. Addressing computational noise showed that further research is warranted to better model the initial proportion used for the species. Combining this point with the additional constraint suggested in Section 9.1.1 should reduce the number of generations required to converge. Even better, it may prove useful in facilitating a better solution rate for the algorithm(s). Table 13 details the puzzles evaluated in this research. Puzzles not cited will be made available online.

Table 13. *Puzzles Used in the Evaluation of (p)RFS (* Indicates puzzles solved by pRFS and not by RFS)*

Sudoku Size	Puzzle Name	Solved
3	9x9codeOriginal	Yes
3	sudokugeantG9_171aEasy (Giant Sudoku, 2016)	Yes
3	sudokugeantG9_171bMedium (Giant Sudoku 2016)	Yes
3	sudokugeantG9_171cDifficult (Giant Sudoku, 2016)	Yes
3	usaToday1Easy (*USA Today*, 2008)	Yes
3	usaToday2Easy (*USA Today*, 2008)	Yes
3	usaToday3Easy (*USA Today*, 2008)	Yes
3	usaToday4Easy (*USA Today*, 2008)	Yes
3	usaToday5Easy (*USA Today*, 2008)	Yes
3	usaToday6Easy (*USA Today*, 2008)	Yes
3	usaToday7Easy (*USA Today*, 2008)	Yes
3	usaToday8Easy (*USA Today*, 2008)	Yes
3	usaToday9Easy (*USA Today*, 2008)	Yes
3	usaToday10Easy (*USA Today*, 2008)	Yes
3	usaToday11Easy (*USA Today*, 2008)	Yes
3	usaToday12Easy (*USA Today*, 2008)	Yes
3	usaToday13Easy (*USA Today*, 2008)	Yes
3	usaToday14Easy (*USA Today*, 2008)	Yes
3	usaToday15Easy (*USA Today*, 2008)	Yes
3	usaToday16Easy (*USA Today*, 2008)	Yes
3	usaToday41Easy (*USA Today*, 2008)	Yes
3	usaToday42Easy (*USA Today*, 2008)	Yes
3	usaToday43Easy (*USA Today*, 2008)	Yes
3	usaToday44Easy (*USA Today*, 2008)	Yes
3	usaToday45Easy (*USA Today*, 2008)	Yes
3	usaToday46Easy (*USA Today*, 2008)	Yes
3	usaToday47Easy (*USA Today*, 2008)	Yes
3	usaToday48Easy (*USA Today*, 2008)	Yes
3	9x9Hard	No
3	sudokugeantG9_171dEvil (Giant Sudoku, 2016)	No
4	PuzzleMagazineFree1	Yes
4	sudokugeantG16_171Medium (Giant Sudoku, 2016)	Yes
4	16x16_1	No
4	16x16_2	No
4	KDB23N1Int	No
5	25x25sudoku_magazine_and_GECCO_LBA	Yes
5	PuzzleMagazineFree2*	Yes
5	sudokugeantG25_171Evil (Giant Sudoku, 2016)	No
6	Puzzle-Book	Yes
6	sudokugeantG36_171Easy (Giant Sudoku, 2016)	Yes
6	sudokugeantG36_171Difficult (Giant Sudoku 2016)	No
7	sudokugeantG49_170Medium (Giant Sudoku, 2016)	No

Table 13. Continued

8	sudokugeantG64_171Easy (Giant Sudoku 2016)	No
9	sudokugeantG81_168Easy (Giant Sudoku 2016)	No
10	sudokugeantG100_169Evil (Giant Sudoku, 2016)	No
15	enjoysudoku225	No

REFERENCES

Boole and Partners. (2021). *OptiNest* (Version No. 2.3.1.). [Computer software]. Boole and Partners. www.boole.eu 18

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs Paralleles*, *10*(2), 141-71. 15

Dam, S., Mandal, G., Dasgupta, K., & Dutta, P. (2015). Genetic algorithm and gravitational emulation based hybrid load balancing strategy in cloud computing. *Proceedings of the 2015 Third International Conference on Computer, Communication, Control and Information Technology (C3IT)*, 1–7. doi=10.1109/C3IT.2015.7060176 20

Delcam Plc. (n.d.). *ArtCam Insignia* (Version 3.000k [discontinued]). [Computer software]. Delcam. www.artcam.com. 18

Dunlop, D., Varrette, S., & Bouvry, P. (2008). On the use of a genetic algorithm in high performance computer benchmark tuning. *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, *SPECTS 2008*, Art. No. 4667550, 105-113. 20

Fogel, L. J., & Burgin, G. H. (1969). Competitive goal-seeking through evolutionary programming. *Contract Air Force, 19*(628), 5927. Air Force Cambridge Research Laboratories. 13

Giant Sudoku and X-Sudoku. (n.d.). *Sudokugeant*. https://sudokugeant.cabanova.com/ 76

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Professional. 14

Goldberg, D. E., & Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithmsand Their Application*, 41–49. L. Erlbaum Associates Inc. 15

Gu, J., Li, N., Tan, Q., & Wei, W. (2007). A novel niche genetic algorithm with local search ability. *2007 IEEE Congress on Evolutionary Computation*, 4606–4609. doi=10.1109/CEC.2007.4425075 2, 7

Harris, M. (2008). *Optimizing parallel reduction in CUDA* [Webinar]. NVIDIA. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf 56

Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applicationsto biology, control, and artificial intelligence.* University of Michigan Press. https://books.google.com/books?id=JE5RAAAAMAAJ 13, 14

Horn, J. (1997). *The nature of niching: Genetic algorithms and the evolution of optimal, cooperative populations*. (Publication No. 97008). [Master's Thesis, Univ. of Illinois]. Illinois Genetic Algorithms Laboratory. 15

Horn, J. (2002). Resource-based fitness sharing. In J. J. Merelo Guervós, P. Adamidis, H. Beyer, J. L. Fernández-Villacañas Martín, & H. Schwefel (Eds.). *Parallel Problem Solving from Nature—PPSN VII: 7th International Conference, Granada, Spain, September 7-11, 2002, Proceedings* (pp. 381–390). Springer. https://doi.org/10.1007/3-540-45712-7_37 7, 16

Horn, J. (2005). Coevolving species for shape nesting. *2005 IEEE Congress on Evolutionary Computation*, *2*, 1800-1807. doi: 10.1109/CEC.2005.1554906 16, 17, 19

Horn, J. (2007a). Optimal nesting of species for exact cover of resources: Two against one. *Proceedings of the IEEE Symposium on Foundations of Computational Intelligence, FOCI 2007, Honolulu, Hawaii, 1-5 April 2007* (pp. 322–330). https://doi.org/10.1109/FOCI.2007.372187 18

Horn, J. (2007b). Optimal nesting of species for exact cover of resources: Two against many. In H. Lipson (Ed.), *Genetic and Evolutionary Computation Conference, GECCO 2007,*

Proceedings, London, England, UK, July 7-11, 2007 (pp. 448–455). https://doi.org/10.1145/1276958.1277056 18

Horn, J. (2010). Pure co-evolution for shape nesting. *Proceedings of the International Conference on Evolutionary Computation* (Vol. 1), 255-260. IJCCI. DOI: 10.5220/0003089402550260 6, 18, 19

Horn, J. (2013). Np-completeness and the coevolution of exact set covers. *GECCO 2013*, 1597 – 1604. Association for Computing Machinery. https://doi.org/10.1145/2463372.2465807 21

Horn, J. (2014). Co-evolution of sudoku solutions. *Mendel*, 117–122. 6, 16, 21, 25, 43

Horn, J. (2017). Solving a large sudoku by co-evolving numerals. *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2017*, 29–30. Association for Computing Machinery. https://doi.org/10.1145/3067695.3082050 21

Horn, J. (2022). Deep-Scale Evolution for industrial shape nesting. Michigan Strategic Fund Grant Agreement Ref. CASE 335618, Subcontract WSU 22109. 23

Horn, J., & Goldberg, D. E. (1998). A timing analysis of convergence to fitness sharing equilibrium. In A. E. Eiben, T. Bäck, M. Schoenauer, & H. Schwefel (Eds.), *Parallel Problem Solving from Nature—PPSN V* (pp. 23–33). Springer. 75

Horn, J., Goldberg, D. E., & Deb, K. (1994). Implicit niching in a learning classifier system: Nature's way. *Evolutionary Computation*, *2*(1), 37–66. https://doi.org/10.1162/evco.1994.2.1.37 14, 15

Kaliappan, M., Augustine, S., & Paramasivan, B. (2016). Enhancing energy efficiency and load balancing in mobile ad hoc network using dynamic genetic algorithms. *Journal of Network and Computer Applications, 73*(C), 35–43. https://doi.org/10.1016/j.jnca.2016.07.003 20

Kumar, V., Grama, A. Y., & Vempaty, N. R. (1994). Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing, 22*(1), 60–79. https://doi.org/10.1006/jpdc.1994.1070 32, 33

Lynce, I., & Ouaknine, J. (2006). Sudoku as a SAT problem. *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale, Florida*. 21

Marco, N., & Lanteri, S. (2000). A two-level parallelization strategy for genetic algorithms applied to optimumshape design. *Parallel Computing*, *26*(4), 377–397. https://www.sciencedirect.com/science/article/pii/S0167819199001167 20

Monk, J., Hanselman, K., King, R., Flagg, R., Zhu, Y., & Segee, B. (2012). Solving sudoku using particle swarm optimization on CUDA. *Proceedings of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications.* Semantic Scholar. https://www.semanticscholar.org/paper/Solving-Sudoku-using-Particle-Swarm-Optimization-on-Monk-Hanselman/dc7e42292d9c71dd6787c3832b38eca1f198157b 21

MTC Software. (n.d.). *ProNest* (Version 8.2.1.1 [discontinued]). [Computer software]. Hypertherm. https://www.hypertherm.com/en-US/hypertherm/pronest/pronest-cadcam-nesting-software/. 18

Mühlenbein, H. (1989). Parallel genetic algorithms, population genetics and combinatorial optimization. *Proceedings of the Third International Conference on Genetic Algorithms*, 416–421. Morgan Kaufmann Publishers Inc. 15

Nicolau, M., & Ryan, C. (2006). Solving sudoku with the gauge system. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, & A. Ekárt (Eds.), *Genetic Programming. EuroGP 2006. Lecture Notes in Computer Science*, *3905*, 213 – 224. Springer. https://doi.org/10.1007/11729976_19213–224. 21

NVIDIA, Vingelmann, P., & Fitzek, F. H. P. (2020). CUDA, release: 10.2.89. Retrieved from https://developer.nvidia.com/cuda-toolkit 6, 29, 44

Perry, Z. A. (1984). *Experimental study of speciation in ecological niche theory using genetic algorithms.* [Doctoral dissertation, University of Michigan]. ProQuest LLC. http://libproxy.library.wmich.edu/login?url=https://www.proquest.com/dissertations-theses/experimental-study-speciation-ecological-niche/docview/303295516/se-2?accountid=15099 14

Puzzles used in the evaluation of (p)RFS. (2022). GitHub. Retrieved from https://github.com/blaynear/Sudoku_Puzzles.git 75

Rechenberg, I. (1973). *Evolutions strategie: Optimierung technischer systeme nach prinzipien der biologischen evolution* (Problemata 15). Stuttgart-Bad Cannstatt. https://www.bibsonomy.org/bibtex/2f051691c05943d7b006e71b121b4e53e/danfunky 13

Rogers, B. (2017). *CUDA sudoku: A massively parallel implementation of an inherently parallel algorithm*. [Undergraduate senior project, Northern Michigan University]. 8

Romero, M., & Urra, R. (2022). CUDA Memory Heirarchy diagram. http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm

Sato, Y., Hasegawa, N., & Sato, M. (2011). Acceleration of genetic algorithms for sudoku solution on many-core processors. *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. 20

Schwefel, H. P. (1974). *Numerische optimierung von computer—Modellen*. [Doctoral dissertation, Technical University, Germany]. Reprinted by Birkhäuser (1977). 13

Simonis, H. (2005). Sudoku as a constraint problem. In B. Hnich, P. Prosser, & B. Smith (Eds.), *Proceedings of the 4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems*, 13–27. 21

Tsutsui, S., & Collet, P. (Eds.). (2013). *Massively parallel evolutionary computation on GPGPUs* (Natural Computing Series). Springer. https://doi.org/10.1007/978-3-642-37959-8 20

USA Today. (2008). *USA Today: Sudoku, 200 puzzles for all levels* (#22). Multi Media International Inc. 76

Wahib, M., Munawar, A., Munetomo, M., & Akama, K. (2011). Optimization of parallel genetic algorithms for NVIDIA GPUs. *2011 IEEE Congress of Evolutionary Computation (CEC)*, 803– 811. doi=10.1109/CEC.2011.5949701 20

Wang, T., Liu, Z., Chen, Y., Xu, Y., & Dai, X. (2014). Load balancing task scheduling based on genetic algorithm in cloud computing. *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 146–152. doi=10.1109/DASC.2014.35 20

Zomaya, A. Y., & Teh, Y. (2001). Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems, 12*(9), 899 – 911. doi=10.1109/71.954620 20