Building N Birds With 1 Store: Parallel Simulations of Stochastic Evolutionary Processes

Building n Birds with 1 Store:

Parallel Simulations of Stochastic Evolutionary Processes

Billy Janitsch

A thesis presented to Computer Science
in partial fulfillment of the honors requirements
for the degree of Bachelor of Arts

Harvard College
Cambridge, Massachusetts

April 1, 2015

Abstract

Stochastic processes are used to study the dynamics of evolution in finite, structured populations. Simulations of such processes provide a useful tool for their study, but are currently limited by computational speed and memory bottlenecks, even when naively parallelized. This thesis proposes two novel parallelization methods for simulating a particular class of evolutionary processes known as "games on graphs." The theoretical speed-up and scalability of these methods is analyzed across various parameters. A novel approximate parallel method is also proposed, which allows for further speed-up at the expense of some accuracy. Discussion of implementation considerations follows, and a resulting implementation in Python is used to provide empirical performance results which match closely with theoretical ones. Applications are suggested for a variety of open problems in biology, behavioral economics, political science, and linguistics.

Acknowledgments

To David Parkes, my advisor, to Martin Nowak and Yaron Singer, my readers, and to Stephen Chong, for their unwavering enthusiasm, encouragement, and kindness, and for collectively teaching three of the most inspiring classes I've taken as an undergraduate.

To everyone, past and present, at the Program for Evolutionary Dynamics, including Ben Adlam, Erez Lieberman Aiden, Oliver Hauser, Alison Hill, Moshe Hoffman, Jean-Baptiste Michel, Alex Peysakhovich, Dave Rand, Carl Veller, and Erez Yoeli.

To my roommates and friends, for their support and persistent reminders to eat, sleep, and shower.

Finally, to my mother, for her love, and for everything else.

Contents

Chapter 1

Introduction

1.1 Evolutionary Processes and Simulations

Beginning midway through the 20th century with the modern synthesis [1], the mathematical study of evolution has grown to unite biology, economics, mathematics, and computer science. Theoretical models allow us to deeply understand the evolutionary processes that govern our world and give rise to an incredible diversity of traits, structures, and behaviors. Such models have been applied to domains as far-reaching as sociology, psychology, linguistics, medicine, and political science [16].

Evolutionary models exist in both deterministic and stochastic forms. While deterministic models are typically more tractable to analyze, stochastic models more accurately represent inherent randomness involved in biological and other contexts. Indeed, many significant biological phenomena such as neutral drift arise exclusively in stochastic formulations [8].

For stochastic models which are complex enough to evade deep mathematical analysis, researchers often turn to computational simulations to understand their dynamics and explore conjectures [2, 4, 17]. In particular, for evolutionary models featuring large populations with complex and irregular structure (such as those involving social networks), mathematical results remain fairly rudimentary, and simulations provide a promising approach [12, 13].

However, such simulations often have significant computational limits. They require maintaining large, complex states, and performing immense numbers of computationally difficult calculations that scale poorly with model parameters such as population size. While some work has been done to invent more efficient algorithms to perform such calculations, even the most efficient single-processor implementations of stochastic simulations can be very slow.

As a result, researchers without access to supercomputers are frequently forced to restrict the scope of their simulations, working with small population sizes, simple network structures, and simplified models [12, 13, 4, 17, 15]. This not only leads to uncertainty about the applicability and generalizability of their results, but also prevents many simulations from approaching the scale of real-world relevancy, such as those on large-scale social networks.

Parallel computing provides a promising approach to solve this problem. Indeed, simulation experiments are typically parallelized in the sense that multiple independent simulations are run simultaneously on multiple processors. However, this results in a linear increase in memory usage, which is a massive problem when individual simulations already push the limits of what can be stored in memory.

Less coarse parallelism which does not suffer from this memory explosion constitutes a much harder problem, especially in light of the non-deterministic nature of the processes. This thesis proposes, implements, and tests novel methods of parallelizing individual simulations of a particular class of processes called “games on graphs”.

1.2 Related Work

The literature for parallel computing in general is extremely rich. The main limitation of this research as applied to stochastic processes is that it deals largely with parallel approaches to entirely deterministic processes and algorithms [6], and is therefore often incapable of handling the non-determinism present in evolutionary models. However, the stochastic processes of interest typically involve the computation of large deterministic subproblems (e.g. weighted random selection [14]), which often have been more widely studied. Thus, I will often defer to existing results and algorithms for such problems.

Parallel computing also provides an approach to reason about computational runtime and speed-up as a function of how many processors are available to execute computations in parallel. While the standard model similarly suffers from assuming various kinds of determinism, it is nevertheless used as a base for an extended, non-deterministic model.

The literature for evolutionary dynamics, and for games on graphs in particular, is quite extensive as well. However, while research in the field regularly employs simulation methods, published work almost exclusively focuses on the results of such simulations, rather than on the details of their computational implementation [12, 13, 4, 17]. Thus, the evolutionary dynamics literature will be primarily used to define the problem space. Specifically, recent work will be reviewed to formulate a general class of stochastic processes capable of representing a wide variety of evolutionary phenomena. Open questions in the literature will also be used to suggest applications of the parallelization methods.

Despite the relative size of each field individually, their intersection is relatively sparse. In particular, there does not appear to be any published work on parallelizing simulations of this particular class of evolutionary models, and there is limited work on parallelizing stochastic processes in general [3].

1.3 Outline

In Chapter 2, I introduce the general form of stochastic evolutionary process that I will analyze. This model is a synthesis of recent work in the field, generalized to allow for a wide variety of specific implementations. I review some of the canonical results obtained from varieties of this model as well as open questions in this domain.

In Chapter 3, I introduce a theoretical model of non-deterministic parallelism and apply it to various classes of evolutionary processes. I propose two main parallelization methods, and investigate the extent to which they are applicable across different types of models. Their theoretical speed-up and scalability are analyzed.

In Chapter 4, I extend this model to allow for parallelization approaches which only approximate, rather than exactly replicate, the dynamics of their respective sequential stochastic processes. I discuss their divergence, analyze the theoretical trade-off between speed-up and accuracy, and review the relevance of accuracy to various models.

In Chapter 5, I discuss the implementation of the resulting parallel simulation methods, considering trade-offs between various data structures and algorithms in terms of complexity and runtime. This chapter accompanies an actual implementation in the form of a Python library capable of running both sequential and parallel simulations.

In Chapter 6, this implementation is used to provide empirical performance benchmarks across a variety of experiments, testing the roles of various forms and parameters of the general stochastic model. These results are compared with the theoretical results obtained in Chapters 3 and 4.

Finally, in Chapter 7, I conclude by suggesting extensions and applications of the parallel approaches to open problems in biology, behavioral economics, political science, and linguistics.

Chapter 2

Evolutionary Processes

2.1 Population Structure

I consider a general class of *evolutionary processes* consisting of a fixed-size *population* of agents which evolves over time according to particular implementations of selection and mutation. The model defines a *population structure*, which does not change over time[1], and a *population composition*, which does. The process's *update rule* defines the stochastic dynamics of the population composition over time.

The *population structure* describes the set of agents in the population and the connections between them. Depending on the context of the model, agents are generally thought of as biological organisms, but may also represent institutions. Context also informs the interpretation of inter-agent connections, which may represent spatial/geographical organization of the population, semantic relationships among the population (as in a human social network in which a link represents a friendship), or channels through which information can flow. The connections are used as a proxy for the relative likelihood of interaction between two agents.

[1]There has been some work involving dynamic populations where agents can enter or exit the population and the connections between them can change over time [10]. Unfortunately, this complicates existing analysis significantly, and such work is currently very rudimentary. Therefore, I restrict my analysis to static, fixed-size population structures.

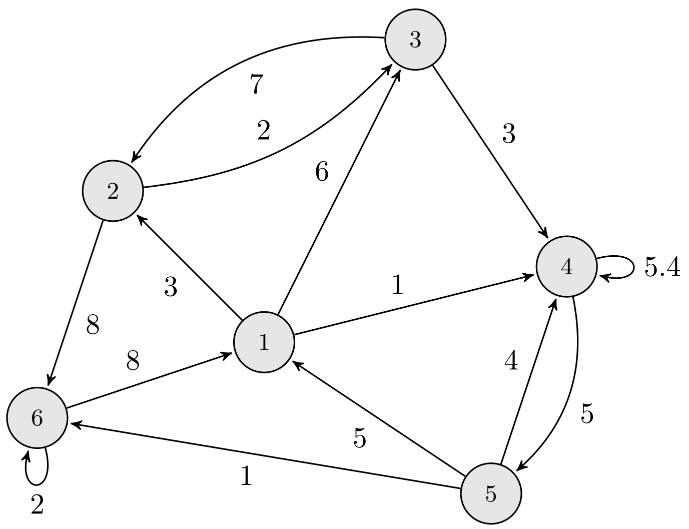


Figure 2.1: An example population structure, with agents $N = \{1, 2, 3, 4, 5, 6\}$ and non-normalized edge weights. Note that agents 4 and 6 are neighbors of themselves, and (2, 3) and (4, 5) are the only pairs of mutual neighbors. The 1st in-neighborhood of agent 1 is $\Psi(1) = \{5, 6\}$, and the 2nd out-neighborhood of agent 2 is $\Omega^2(2) = \{1, 2, 4, 6\}$.

Formally, the structure is given by a weighted, directed graph $G = (N, E, w)$. N is set of nodes $\{1, 2, \dots\}$ representing agents, with $|N| \geq 2$, connected by edges $(i, j) \in E$ where $i, j \in N$, with edge weighting function $w : N \times N \to \mathbb{R}_0^+$. Let $w(i, j) = 0$ iff $(i, j) \notin E$ as a convenience. Outgoing edges from each node have weights which are typically normalized such that

$$\sum_{j \in N} w(i, j) = 1$$

yielding a right stochastic adjacency matrix. Finally, assert that G is weakly connected (i.e. $\forall i, j \in N$. it is possible to reach node j from node i by traversing edges in either direction). This assertion prevents us from working with graphs that could more easily be thought of as entirely separate populations, where each population is a single weakly connected component. Moreover, it ensures that every agent has at least one neighbor.

Agent j is a *neighbor* of agent i if $(i, j) \in E$. This relationship is not bidirectional in general, and it is possible for an agent to be a neighbor of itself. However, agent i and j are *mutual neighbors* of each other if $(i, j) \in E$ and $(j, i) \in E$. Let $\Omega(S)$ and

$\Psi(S)$ represent the 1st *out-neighborhood* and *in-neighborhood* of a set of nodes $S \subseteq N$, respectively. These are defined as follows

$$\Omega(S) = \bigcup_{i \in S} \{ j \in N \mid (i,j) \in E \}$$
$$\Psi(S) = \bigcup_{i \in S} \{ j \in N \mid (j,i) \in E \}$$

This notation is overloaded slightly to let $\Omega(i) = \Omega(\{i\})$ and $\Psi(i) = \Psi(\{i\})$ for $i \in N$. Intuitively, the 1st out-neighborhood of i is the set of immediate neighbors of i, and the 1st in-neighborhood of i is the set of agents with i as one of their neighbors. Thus, $|\Omega(i)|$ and $|\Psi(i)|$ are the *out-degree* and *in-degree* of i, respectively. $\Omega^n(S)$ (Ω composed n times) is refered to as the nth out-neighborhood of S, and similarly $\Psi^n(S)$ is the n-th in-neighborhood of S. For examples of this notation, see Figure 2.1.

This weighted digraph is a very general model for population structure, but many specific models enforce tighter constraints. A few special cases of population structure are defined below.

Definition 2.1.1 (Weakly Undirected Population Structure)**.** A *weakly undirected* population structure is one where $\forall (i,j) \in E : (j,i) \in E$.

Definition 2.1.2 (Strongly Undirected Population Structure)**.** A *strongly undirected* population structure is one where $\forall i,j \in N : w(i,j) = w(j,i)$.

A weakly undirected network graph is undirected in the sense that any edge is complemented by an edge with opposite direction. In a strongly undirected network graph, these edges pairs have equal weight and can thus be thought of as a single bidirectional edge. In both types of undirected population structures, neighbor relationships are bidirectional, such that $\forall S \subseteq N : \Omega(S) = \Psi(S)$.

In practice, population structures are typically at least weakly undirected, and often strongly undirected, since most types of connections between agents are bidirectional, such as physical proximity and societal structure. However, directionality is occasionally used to model varying degrees of influence[2].

Definition 2.1.3 (Unweighted Population Structure)**.** An *unweighted* population structure is one where $\forall i \in N, \exists c$:

$$w(i,j) = \begin{cases} c & \text{if } j \in \Omega(i) \\ 0 & \text{otherwise} \end{cases}$$

All outgoing edges from any single node are of equal weight, though edge weights need not be constant across the entire graph. As will be seen, weights will only ever be compared between outgoing edges of the same node, so this is equivalent to conventional notions of an unweighted graph.

Definition 2.1.4 (Well-mixed Population Structure)**.** A *well-mixed population* is one where $\exists c : \forall i, j \in N : w(i,j) = c$.

Well-mixed populations are both strongly undirected and unweighted. In such a population, all agents are equally connected to all other agents, including themselves, so it is equally likely for any agent to interact with any other agent. Such populations can be thought of as lacking structure entirely. This notion of "amount of structure" is captured by *density*, which is the ratio of the number of edges that exist to the number of possible edges, or $\frac{|E|}{|N|^2}$. Intuitively, a low density corresponds to a high amount of structure.

Population structure can also be analyzed in terms of its degree distribution function $\Lambda(n)$, which is the probability that a randomly selected agent has exactly n neighbors.

[2]Consider a population where nodes represent either politicians or citizens, and assume that citizens can influence each other equally, whereas politicians can influence citizens but are not influenced by them. In such a model, directional edges run from politicians to citizens but not vice versa, yielding a directed population structure.

Formally,

$$\Lambda(n) = \frac{\left|\left\{i \in N \mid \Omega(i) = n\right\}\right|}{|N|}$$

There are several canonical methods to randomly generate population structures for a given population size $|N|$. Often, these are designed such that their resulting graphs exhibit specific properties which have been empirically observed in real-world population networks. Such population generation procedures are useful when attempting to determine whether some property is generally true across a particular kind of population structure, which can otherwise be difficult given a small sample size of real-world data. A few particularly important generation procedures are reviewed below.

Definition 2.1.5 (Independent Edge Formation)**.** A random population structure with independent edge formation is generated according to the following process with parameter $p \in (0, 1)$. For some population size $|N|$, iterating over all $i, j \in N$, edge (i, j) is created with independently sampled probability p.

A population generated using this method has a binomial degree distribution and an expected density of p. Such populations are generally very different from real-world populations in terms of topology. In particular, they fail to exhibit certain well-observed properties of social networks, including "small worlds" and "edge clustering". However, their simplicity makes them very tractable for mathematical analysis.

Definition 2.1.6 (Preferential Attachment)**.** A random, unweighted, undirected population structure with preferential attachment is generated as follows with parameter $m \in \mathbb{N}^+$. Beginning with a complete graph of m agents, $|N| - m$ agents enter the population one-by-one. Each agent forms m edges with agents in the population, sampled without replacement proportional to current degree.

Definition 2.1.7 (Copying)**.** A random, unweighted population structure with copying is generated by the following procedure with parameters $m_r, m_c \in \mathbb{N}^+$, and $p \in (0, 1)$. Beginning with a complete graph of $1+m_r+m_c$ agents, $|N|-1-m_r-m_c$ agents enter the population one-by-one. Each agent i samples m_c agents uniformly from the population, and forms an edge toward them with independently sampled probability p. Then, it samples m_r agents from $\Omega^2(i)$, and forms an edge toward them with independently sampled probability p.

The copying model produces population structures which exhibit very similar properties to empirically-observed social networks. It will be used to generate populations used for experiments in Chapter 6.

2.2 Population Composition and Dynamics

The *population composition* describes the state of the population over discrete units of time $t \in \{0, 1, \dots\}$. There is a (finite or infinite) set of *types* $\mathbf{Q} = \{q_1, q_2, \dots\}$, and each agent has a single type which can change over time. Depending on the context of the model, these types can be interpreted in a variety of ways, for instance as biological phenotypes, strategies played by agents in games, or sets of ideas of beliefs that determine behavior. Likewise, an agent's transition between two types can be contextually interpreted, for instance as reproduction, imitation, or conversion of belief.

Formally, the state of the population at a given time is represented as a vector

$$s = \langle s_1, \dots, s_{|N|} \rangle$$

where s_i represents the type of agent $i \in N$. If s is the population state at time t, s' is generally used to represent the population state at time $t+1$. Let $s[i \mapsto x]$ represent the vector equivalent to s except at position i where s_i has been replaced by x. For example, if $s = \langle q_1, q_2, q_3, q_4 \rangle$, then $s[2 \mapsto s_4] = \langle q_1, q_4, q_3, q_4 \rangle$. The initial population state must be provided, and may be random or fixed.

The interaction function $\sigma : \mathbf{Q} \times \mathbf{Q} \to \mathbb{R}_0^+$ defines the payoff that an agent of type $q_i \in \mathbf{Q}$ receives from interacting with an agent of type $q_j \in \mathbf{Q}$. In population state s, an agent $i \in N$ has *fitness*

$$f_i = \sum_{j \in \Omega(i)} w(i,j)\sigma(s_i, s_j)$$

which is a weighted average of interaction payoffs with its neighbors in their current states. Note that because of the population structure's normalized edge weights, fitness is not biased towards agents with more neighbors. Further, fitness is bounded by some maximum fitness $f_{\max}$ [16]. $q_i \in \mathbf{Q}$ has *constant* fitness c if $\forall q_j \in \mathbf{Q} : \sigma(q_i, q_j) = c$, i.e. if its interaction payoff does not depend on the type of the agent that it is interacting with. Total population fitness, the sum of the fitnesses of all agents, is formally defined as

$$f = \sum_{i \in N} f_i$$

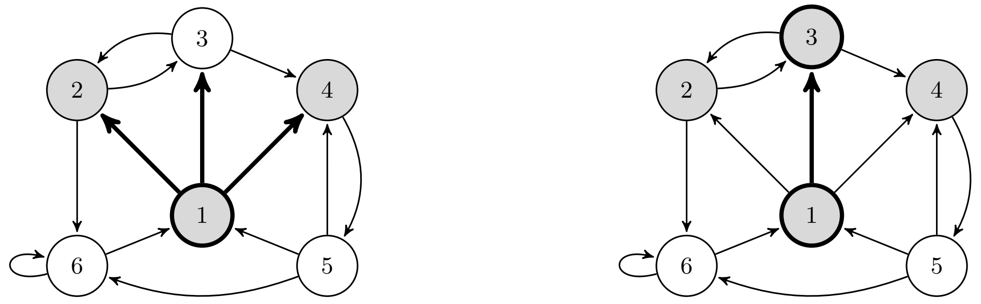
Finally, there is an *update rule* which specifies the transition probability from s to s'. The only formal restriction on the update rule is that it must exhibit the *Markov property.*

Definition 2.2.1 (Markov Property)**.** Let random variable S_t represent the state of the population at time t. An evolutionary process exhibits the Markov property if

$$P(S_{t+1} = s_{t+1} \mid S_t = s_t) = P(S_{t+1} = s_{t+1} \mid S_t = s_t, \ldots, S_1 = s_1)$$

Informally, this means that s', given s, does not depend on any state prior to s. Update rules typically implement selection, and occasionally mutation. Several canonical classes of update rules are defined below. They will be the main feature by which processes are distinguished in the context of parallelization methods introduced in Chapter 3, since they turn out to exhibit widely differing computational properties.

Definition 2.2.2 (Birth-Death Process)**.** A *birth-death process* **BD** has the following update rule. First, an agent $r \in N$ is chosen to reproduce by a random selection from the entire population proportional to each agent's fitness f_r, with non-normalized PMF



(a) Agent 1 is selected to reproduce. It will replace one of its neighbors $\Omega(1)$.

(b) Agent 3 is selected to die. It takes on the type of reproducing agent 1.

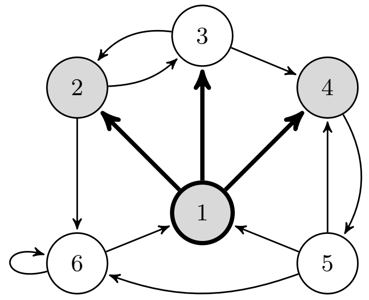
Figure 2.2: An example of one round of **BD** on an unweighted population with type space $\mathbf{Q} = \{W, G\}$, represented by white and grey shading of the agents, respectively.

$p_R(r) = f_r$. Next, one of its neighbors $d \in \Omega(r)$ is chosen to die proportional to edge weight $w(r, d)$, with non-normalized PMF $p_D(d \mid r) = w(r, i)$. The reproducing agent replaces the dying one, yielding $s' = s[d \mapsto s_r]$.

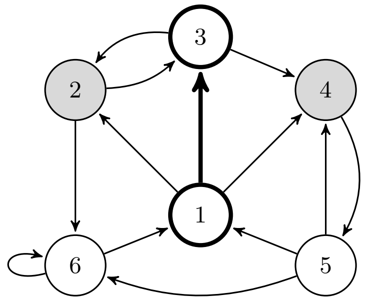
For an example, see Figure 2.2. **BD** implements selection by favoring the reproduction of fitter agents. Note that it is possible to sample $d = r$, such that an agent replaces itself, if $r \in \Omega(r)$.

Definition 2.2.3 (Death-Birth Process)**.** A *death-birth process* **DB** has the following update rule. First, an agent $d \in N$ is chosen to die by a uniformly random selection from the entire population, i.e. sampled from a distribution with PMF $p_D(d) = \frac{1}{|N|}$. Next, one of its neighbors $r \in \Omega(d)$ is chosen to reproduce proportional to edge weight and fitness, i.e. sampled from a distribution with non-normalized PMF $p_R(r \mid d) = w(d, r) f_r$. The reproducing agent replaces the dying one, yielding $s' = s[d \mapsto s_r]$.

For an example, see Figure 2.3. **DB** can also be thought of as an imitation process, whereby a randomly chosen agent decides to imitate the type of one of its neighbors proportional to their fitness and edge weight. Note that, as in **BD**, it is possible for an agent to replace itself. **BD** and **DB** are equivalent in a well-mixed population, but their dynamics differ subtly in other kinds of population structures [8].



(a) Agent 1 is selected to die. It will be replaced by one of its neighbors $\Omega(1)$.



(b) Agent 3 is selected to reproduce. Its type is taken on by agent 1.

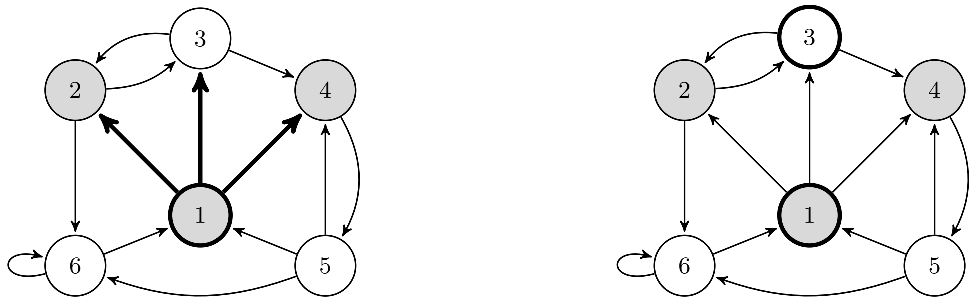
Figure 2.3: An example of one round of **DB**.

Definition 2.2.4 (Pairwise-Comparison Process)**.** A *pairwise-comparison process* **PC** has the following update rule. As in **DB**, an agent $d \in N$ is chosen to potentially die by a uniformly random selection from the entire population. Next, one of its neighbors $r \in \Omega(d)$ is chosen to *potentially* reproduce proportional to edge weight $w(d, r)$, with non-normalized PMF $p_R(r \mid d) = w(d, r)$. Next, with probability $\theta(f_r - f_d)$, where $\theta : \mathbb{R} \to [0, 1]$ is some monotonically increasing function, the reproducing agent successfully replaces the dying one yielding $s' = s[d \mapsto s_r]$. Otherwise, $s' = s$.

For an example, see Figure 2.4. Examples of θ include $\theta(\Delta) = \max(0, \frac{\Delta}{f_{\max}})$, where $f_{\max}$ is the maximum possible fitness. This lets an agent update its strategy only if doing so would yield a higher fitness for that agent. **DB** is a special case of **PC** with $\theta(\Delta) = 1$, but as will be seen, it is worth considering as its own process due to computational differences resulting from its lack of dependence on Δ.

Definition 2.2.5 (Best Response Process)**.** A *best response* process (**BR**) has the following update rule. An agent $d \in N$ is chosen to update by a uniformly random selection from the population. It updates its type to the $q \in \mathbf{Q}$ which maximizes its fitness f'_d given the current types of its neighbors $\left\{s_j \mid j \in \Omega(i)\right\}$.

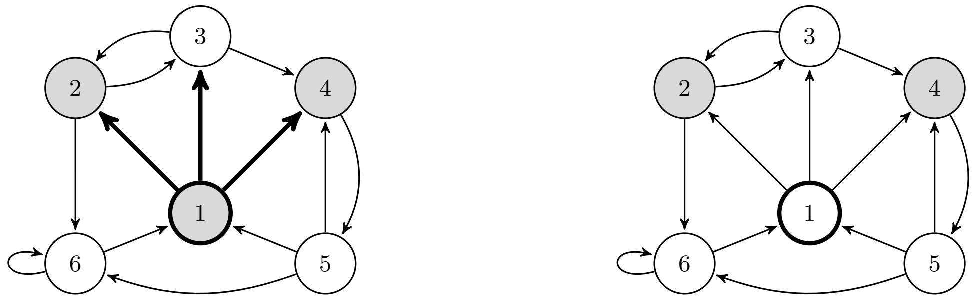
Note that **BR** is *innovative* in the sense that it is capable of introducing a type $q \in \mathbf{Q}$ not currently present in the population state s, unlike the processes described



(a) Agent 1 is selected to potentially die. It may be replaced by one of its neighbors $\Omega(1)$.

(b) Agent 3 is selected to potentially reproduce. However, the reproduction is unsuccessful, so the state of agent 1 does not change.

Figure 2.4: An example of one round of **PC**.



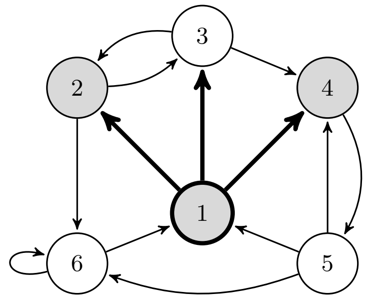
(a) Agent 1 is selected to update. It will choose a new type by considering a best response to its neighbors.

(b) Agent 1 updates its type to optimize its payoff.

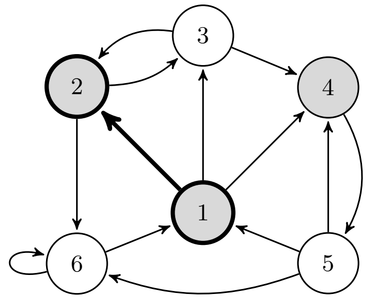
Figure 2.5: An example of one round of **BR**.

above. It is also unique in that, although the selection of d is non-deterministic, the updating agent has a deterministic rule to update its type.

Definition 2.2.6 (Birth-Death with Payoff Affecting Death Process)**.** A *birth-death with payoff affecting death process* **BDD** has the following update rule. First, an agent $r \in N$ is chosen to reproduce by a uniformly random selection from the entire population. Next, one of its neighbors $d \in \Omega(r)$ is chosen to reproduce proportional to edge weight $w(r, i)$ and inverse fitness $\frac{1}{f_i}$, with non-normalized PMF $p_D\left(d \mid r\right) = \frac{w(r,d)}{f_d}$. Finally, the reproducing agent replaces the dying one yielding $s' = s[d \mapsto s_r]$.

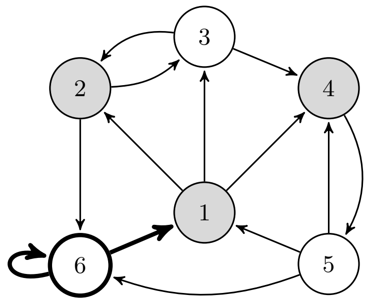


(a) Agent 1 is selected to reproduce. It will replace one of its neighbors $\Omega(1)$.

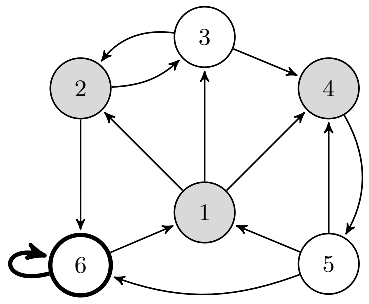


(b) Agent 2 is selected to die. It obtains the type of agent 1 (which happens to be the same as its previous type).

Figure 2.6: An example of one round of **BDD**.



(a) Agent 6 is selected to reproduce. It will replace one of its neighbors $\Omega(1)$.



(b) Agent 6 (the same agent that was selected to reproduce) is selected to die.

Figure 2.7: An example of one round of **DBD**.

Definition 2.2.7 (Death-Birth with Payoff Affecting Death Process)**.** A *death-birth with payoff affecting death process* **DBD** has the following update rule. First, an agent $d \in N$ is chosen to die by a random selection from the entire population weighted by inverse fitness, with non-normalized PMF $p_D(d) = \frac{1}{f_d}$. Next, one of its neighbors $R \in \Omega(d)$ is chosen to reproduce proportional to edge weight, with PMF $p_R(r \mid D = d) = w(d, r)$. Finally, the reproducing agent replaces the dying one yielding $s' = s[d \mapsto s_r]$. Note that it is possible to sample $r = d$, such that an agent may replace itself, if $d \in \Omega(d)$.

Mutation can be introduced into any process as follows. For some constant mutation parameter $\mu \in [0, 1]$, if the result of a round is $s' = s[d \mapsto s_r]$, with probability μ, the result instead becomes $s' = s[d \mapsto x]$, for some uniformly randomly sampled $x \in Q$[3]. Mutation introduces innovation in non-innovative processes, which is useful for avoiding permanent fixation of a single type across the entire population.

Death-first processes can also introduce a *synchronization* parameter $\rho \in [0, 1]$. The idea is to have multiple agents die and be replaced in the same round. Specifically, before updating the state of d, for each agent $i \in N \setminus \{d\}$, i is chosen to die with probability ρ. If it is chosen to die, then its type is updated according to the rule of the process used for d. Thus, ρ allows for a continuous transition between individual update ($\rho = 0$) and full sequencial update, where every agent always updates their type ($\rho = 1$).

2.3 Examples

Below are two examples of models which illustrate the utility of this type of stochastic evolutionary model, as well as the limitations of analytical approaches in studying it, and the limitations of standard inter-simulation parallelization in simulating it.

Moran process

The *Moran process* is one of the earliest and simplest biological stochastic processes, designed to model genetic drift of two alleles in a population with constant selection [8].

Definition 2.3.1. For some $k_A, k_B \in \mathbb{R}^+$, a *Moran process* is a special case of **BD** on a well-mixed population with $\mathbf{Q} = \{A, B\}$, and $\forall q \in \mathbf{Q} : \sigma(A, q) = k_A$ and $\sigma(B, q) = k_B$.

Extensive analytical study of this model has yielded a number of classically beautiful results, including that of its fixation probability, below.

Theorem 2.3.1 (Fixation Probability in a Moran Process)**.** *The fixation probabilities*

[3]A more complex but more realistic implementation of mutation instead uses a weighted selection from the type space, dependent on the current type of the reproducing agent, to model the fact that not all mutations are equally likely.

for types A and B are

$$\Gamma_A(s) = \frac{k_A|A_s|}{k_A|A_s| + k_B|B_s|}$$
$$\Gamma_B(s) = \frac{k_B|B_s|}{k_A|A_s| + k_B|B_s|}$$

where $|q_s|$ is the number of agents with type $q \in \boldsymbol{Q}$ in state s.

Proof. See [8]. □

If $k_A = k_B$, the fixation probability reduces to $\frac{|A_s|}{|N|}$ for A and $\frac{|B_s|}{|N|}$ for B. This is the special case of neutral drift, whereby a neutral mutant present in fraction p of the population will eventually reach fixation with probability p.

In general, this model is mathematically quite tractable due to its simplicity, but its simplifying assumptions constrain its utility. In particular, it is rare to find a real-world population that can be considered well-mixed, since so many factors can bias interaction probabilities.

Prisoner's dilemma on a graph

A more complex model defines interaction payoffs based on the *Prisoner's Dilemma* (PD), a classic model in game theory, on a structured population. This allows for the study of emergence of cooperative behavior as a function of the population's topology.

Specifically, the prisoner's dilemma has type space $\mathbf{Q} = \{C, D\}$ (representing the "strategies" of cooperation and defection, respectively), with interaction function σ given by the following payoff matrix (where the row and the column are the respective arguments of σ):

	C	D
C	2	0
D	3	1

Note that type D has strictly better payoff than C regardless of the type of the other agent, but the overall payoff of a mutual interaction $\sigma(x, y) + \sigma(y, x)$ is maximized when both agents are of type C. Intuitively, cooperation is better for “society”, but defection is better for any individual agent. The general question is whether cooperation can emerge and persist in a population despite the apparent superiority of defection.

While many analytical results have been obtained for various regular population structures such as lattices, rings, and well-mixed populations [7, 1, 11], less progress has been made for large, irregular population structures [1, 12]. One significant result is the $b/c > \langle k \rangle$ rule, which, for a simplified PD, describes the potential for cooperation to emerge in **DB** as a function of the average $\Omega(i)$ and the ratio between the benefit (to others) and cost (to an agent) of cooperation [1, 9]. However, there are a number of open questions concerning this model [16, 1, 2, 12, 13]. For example, it is generally thought that increased population structure promotes cooperation, but this relationship is not well-understood.

Simulations of this model and its variants have been widely employed in current research [12, 5]. However, such simulations have been constrained to relatively small populations in order to compute a statistically valid number of rounds in a reasonable amount of time [12].

This model will be used in Chapter 6 to empirically test the computational speed-up of the parallelization approaches proposed in Chapters 3 and 4.

Chapter 3

Parallelization

3.1 Architecture

I now introduce a framework with which to analyze the theoretical parallelizability of evolutionary processes. I consider two novel methods of parallelization: *intra-round parallelizability* and *inter-round parallelizability*. Informally, the former asks whether, within a single round of a process, there are operations that can be executed in parallel. The latter asks whether multiple rounds of the process can be computed in parallel – that is, whether there are operations that can be computed towards determining s' without having determined s. I analyze the theoretical speed-up of each method across the processes defined in Chapter 2.

It is important to distinguish between theoretical and practical parallelizability; the former makes significant idealistic assumptions about properties of hardware which are unlikely to hold up in practice [6]. Nevertheless, theoretical analysis is a useful and often necessary first step, as its results can generally be translated to practical approaches that retain the same basic properties of scalability, speed, and memory usage [6].

The *parallel architecture* models the hardware on which computations will be run. Suppose there are K *processors*, or *cores*[1], on which computations can be run in parallel.

[1]Technically, the former term is more accurate, but the latter will be used to more apparently

In general, the goal is to maximize the use of these cores over time. It is assumed that each core is equally fast, and that the cores operate independently, such that running one computation on each core is K times as fast as running K computations on one core[2].

The processors each have access to the same shared memory space[3], which can be accessed in unit time. The memory space is modelled as a concurrent-read exclusive-write (CREW) system, whereby multiple processors can read from the same address in memory simultaneously, but only one processor can write to a given address at once. The processors do not share a global clock, so their execution order is non-deterministic.

Let T_K represent the running time of an algorithm given K cores (which is, of course, a function of the input). Then, T_1 is the running time of the sequential algorithm.

Definition 3.1.1 (Speed-up)**.** The computational *speed-up* of a parallel algorithm given K processors is $S_K = \frac{T_1}{T_K}$.

Definition 3.1.2 (Efficiency)**.** The *efficiency* of a parallel algorithm given K processors is $E_K = \frac{S_K}{K}$.

Speed-up and efficiency are used to reason about the effectiveness of parallelization in terms of computation time. $S_K \in O(K)$, referred to as *linear speed-up*, is optimal, representing the case where the algorithm takes K times as long to run on 1 core as it does to run on K cores in expectation. This is equivalent to $E_K = 1$. An efficiency of $E_K \rangle 1$ (superlinear speed-up) is theoretically impossible [6]. In practice, $E_K \langle 1$ (sublinear speed-up) is expected; parallel algorithms generally introduce at least some *parallel*

distinguish this concept from that of evolutionary *processes*.

[2]In reality, neither of these assumptions is likely to be exactly true, and their inaccuracy will vary across hardware. However, averaged across very large numbers of computations, any asymmetry is negligible.

[3]This is refered to as a parallel random-access machine (PRAM) model. Alternatives include a local memory machine model, in which each processor has its own local memory, and a modular memory machine model in which processors request temporary use of memory modules when needed [6]. The PRAM model is the most idealistic, but it is a useful abstraction and can be translated into other models later when designing practical implementations.

overhead, including idle time and extra computational work needed to coordinate parallization. The challenge of parallelization is to determine an algorithm that minimizes this overhead, maximizing E_K.

This means that, in one sense, parallel algorithms are typically strictly *worse* than their sequential counterparts, since $E_K \langle 1$ implies that, in total, more resources are occupied over time for the same result. However, as long as $E_K \geq \frac{1}{K}$, then a parallel algorithm runs more quickly than its sequential counterpart in expectation. Thus, parallelization is useful in situations where computation time is more of a constraint than resource consumption[4].

Typically, E_K decreases in K [6] – a case of diminishing returns, where the effectiveness of increased parallelization plateaus. However, E_K typically increases in input size, informally because more input allows more work to be done synchronously. Thus, different classes of problem sizes are optimized by different degrees of parallelization. In the context of evolutionary processes, problem size most readily corresponds to population size (which in turn influences the number of rounds to simulate). As will be seen, efficiency is also heavily influenced by the type of process and the population structure.

3.2 Task Dependencies

Generally, the simulation of an evolutionary process can be thought of as a set of computations, some of which depend on others. For instance, selecting an agent to reproduce in a round of **BD** depends on having computed the fitness of all agents in the population at the end of the previous around. However, selecting an agent to die in a round of **DB** is completely independent of any other computation. Understanding these relationships between computations allows for reasoning about which computations may

[4]Consider an evolutionary process for which $T_1 = 100$ days in expectation. Supposing, quite reasonably, that a machine with 64 cores is available, even a very low efficiency of $E_{64} = 0.2$ would reduce the running time of the process to under 8 days. In the context of academic research, this trade-off is extremely worthwhile.

occur in parallel and what the resulting speed-ups will be.

Traditional study of parallel algorithms involves representing such relationships as a *task dependency graph* [6], where nodes called *tasks* are discrete computational steps with an associated asymptotic running time called *work*. The outcome of a task is the assignment, or reassignment, of a variable. A directional edge between tasks i and j indicates that i *depends on* j (i.e. that task i must be completed before task j can begin). Another way to think of this is that task j takes the output of task i as input. For example, in **DB**, the computation of the reproducing agent r depends on the computation of the dying agent d, since r must be selected from the neighbors of d. A task dependency graph also has two special tasks, both with work 0: reading input (the initial state of the population) and writing output (the state of the population at the end of every round), which always have only out-edges and in-edges, respectively. The ultimate goal is to compute the output task.

Unfortunately, the randomness involved in stochastic evolutionary processes complicates this type of analysis. Consider **BD**, where $s' = s[d \mapsto s_r]$. s'_i is either remapped to s_r or it is not updated at all (staying equal to s_i), depending on the random selection of d. The task of computing s'_i is dependent on either the task of computing r or the task of computing s_i, but d must be computed first to determine which one. Thus, it is initially unknown whether edges should exist between the task of s'_i and those of s_i and r. Further, the work of a task may also be nondeterministic, such as in the preceding case where the task of computing s'_i has work 0 if it depends on s_i (since the variable is left unchanged) and work 1 if it depends on r (which requires a read from memory).

An overly conservative solution to these problems would be to consider any task fully dependent on all tasks that it might depend on, but introducing unnecessary dependencies has negative consequences for efficiency. Instead, such nondeterministic dependencies can be thought of as being *conditional* on a set of tasks. Once every task in the set has been executed, a conditional dependency can be *resolved*, becoming either a normal

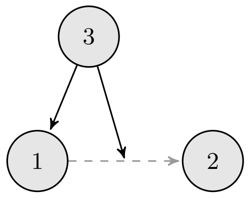


Figure 3.1: Example of a conditional task dependency graph $\langle T, D\rangle$. Tasks $T = \{1, 2, 3\}$ are represented as nodes, and dependencies $D = \{(1, 2), (3, 1), (3, (1, 2))\}$ are represented edges (which can, unusually, point to other edges as well as nodes). Unresolved dependencies are represented as dashed lines. In this case, 3 would need to be computed to determine whether 2 depends on 1, and to compute 1.

dependency or a lack thereof.

Formally, this idea is captured by a *conditional task dependency graph* (CTDG), defined by a vector $\langle T, D\rangle$ consisting of a set of tasks T, and a set of dependencies D. A task $t \in T$ has work δ_t. For $t_1, t_2 \in T$, if t_2 might (or must) depend on t_1, then $(t_1, t_2) \in D$. Similarly, for $t \in T$ and $d \in D$, if d is conditional on t, then $(t, d) \in D$. Thus, dependencies are recursive: $D \subseteq (T \times T) \cup (T \times D)$. See Figure 3.1 for an example.

The set of tasks T contains a set of input variables I, a set of output variables O, and a set of other variables V that must be computed along the way. Since the input variables are always present, they have work 0. Thus, $T = I \cup O \cup V$.

The *total work* in a CTDG is the sum of the work of all of its tasks. This is equal to T_1 [6], since a single processor will, in some order, execute each task exactly once. The *span* is the length of the longest directed path from the input node to the output node, where length is determined by the sum of the work of each node in the path. This represents the sequential bottleneck – the longest sequence of computations that cannot be performed in parallel – and is thus equal to T_∞, the running time given an infinite number of processors [6]. Thus, S_∞ is the ratio of total work to span.

In general, T_K depends on how tasks are *scheduled*, i.e. in what order they are sent to the processors. Thus, to compute T_K, a *scheduler*, which algorithmically determines this ordering on-the-fly, must be specified. Computing the scheduler which results in an optimal, minimized value T^*_K is difficult, especially in the presence of stochastic nondeterminism, but a *greedy scheduler* (which simply sends an idle processor any “available” task) provides a good approximation. Specifically, *any* greedy scheduler results in a T_K within a factor of 2 of the optimal value T_K^*, and converges towards optimal when $\frac{T_1}{K} \gg T_\infty$ or $\frac{T_1}{K} \ll T_\infty$ [6]. Use of a thoughtful, deterministic heuristic can improve performance even further (and allows T_K to be calculated in expectation).

Determining which tasks are available to be sent to a processor at a given time and reasoning about which heuristic the scheduler should use requires a formal notion of *task availability*, which is developed below in the context of a broader model of computation.

A *variable store* σ is used to keep track of computed tasks and their outputs. Formally, $\sigma : T \to *$ is a dynamic function which maps tasks to arbitrary outputs, such that for some task $t \in T$ that has been computed yielding output x, $\sigma(t) = x$. The notation $t \in \sigma$ is used to mean that σ is defined for task t; σ is not defined for tasks which have not yet been computed. Further, let $\sigma[t \mapsto x]$ represent the store equivalent to σ except that t is mapped to x. Finally, let σ_0 represent an empty store which is not defined for any tasks.

A *computational state* on a CTDG $\langle T, D\rangle$ is represented by a vector $\langle \sigma, U, R\rangle$ containing a variable store σ, a set of unresolved dependencies $U \subseteq D$, and a set of resolved dependencies $R \subseteq D$.

The computation function $\pi_t(\sigma, R)$ represents the output of the computation of task $t \in T$, given input $\big\{\sigma(t') \bigm| (t', t) \in R\big\}$. Note that π_t can be non-deterministic; however, it is thought of as having crystallized prior to the computation of the CTDG, such that it is indeed a function as opposed to a relation. The resolution function $\phi_d(\sigma, R)$ represents the resolution of dependency d, given input $\big\{\sigma(t') \bigm| (t', t) \in R\big\}$. This resolution is either $\big\{d\big\}$ or $\emptyset$, depending on whether d resolves to a dependency or not.

There are two ways to make computational progress on a CTDG: by resolving an unresolved dependency, or by computing a task. The former updates the sets of resolved and unresolved dependencies, and the latter updates the variable store. Only available tasks can be computed, and only resolvable dependencies can be resolved.

Definition 3.2.1 (Task Availability)**.** On CTDG $\langle T, D\rangle$ in state $\langle\sigma, U, R\rangle$, task $t \in T$ is *available* if $t \notin \sigma \wedge \nexists(t', t) \in U \wedge \forall(t', t) \in R : t' \in \sigma$.

Definition 3.2.2 (Dependency Resolvability)**.** On CTDG $\langle T, D\rangle$ in state $\langle\sigma, U, R\rangle$, dependency $d \in U$ is *resolvable* if $\nexists(t, d) \in U \wedge \forall(t, d) \in R : t \in \sigma$.

Intuitively, a task is available if it has not already been computed, it has no unresolved dependencies, and all of the tasks it depends on have been computed. An unresolved dependency is resolvable if all of its dependencies are resolved and all of the tasks it depends on have been computed.

Computational progress on a CTDG can be represented by the relation $\rightarrow$, formally captured by the following inference rules, which use task availability and dependency resolution as conditions. $(x, y) \in\rightarrow$ is written using infix notation $x \rightarrow y$ for readability.

$$\textsc{Resolve}\ \frac{d \in U \qquad \nexists(t, d) \in U \qquad \forall(t, d) \in R : t \in \sigma}{\langle\sigma, U, R\rangle \rightarrow \langle\sigma, U \setminus d, R \cup \phi_d(\sigma, R)\rangle}$$

$$\textsc{Compute}\ \frac{t \in T \qquad t \notin \sigma \qquad \forall(t', t) \in R : t' \in \sigma \qquad \nexists(t', t) \in U}{\langle\sigma, U, R\rangle \rightarrow \langle\sigma[t \mapsto \pi_t(\sigma, D)], U, R\rangle}$$

$\rightarrow$ defines the set of actions that can be taken in a given state. Let $\xrightarrow{R}\subseteq\rightarrow$ and $\xrightarrow{C}\subseteq\rightarrow$ represent the subsets of $\rightarrow$ which use only the Resolve and Compute rules, respectively. The notation $\rightarrow^*$ is used to represent the reflexive transitive closure of $\rightarrow$. That is, if state y can be reached from state x by zero or more applications of $\rightarrow$, then $x \rightarrow^* y$. $\xrightarrow{R}^*$ and $\xrightarrow{C}^*$ are similarly defined. Finally, let $\Rightarrow$ represent the relation of "longest computability path", i.e. the longest sequence of computations that can be made from a particular state. Formally, $x \Rightarrow y$ if $x \rightarrow^* y \wedge \nexists z : y \rightarrow z$.

Definition 3.2.3 (Computability)**.** A CTDG $\langle I \cup O \cup V, D\rangle$ is *computable* if $\exists\sigma', U', R' : \langle\sigma_0, \emptyset, D\rangle \rightarrow^* \langle\sigma', U', R'\rangle \wedge \forall t \in O : t \in \sigma'$.

Computability simply ensures that all output tasks can be computed, and is a necessary and sufficient condition for a CTDG to be *valid.*

3.3 Intra-round Parallelization

The potential for intra-round parallelization can be studied by constructing a CTDG $\langle T, D\rangle$ for a single, general round of a process. Across all processes, I is the state of the population at the end of the previous round, including computed fitness, and O is the updated state of the population at the end of the round, also including fitness. V depends on the process, and may include tasks like the selection of an agent to die or reproduce. The dependency set D can be determined in general by reasoning about the update rule of the process.

Example 3.3.1 (CTDG for **BD**)**.** Consider **BD**, which requires selections of an agent to reproduce and an agent to die, so $V = \big\{d, r\big\}$. The selection of r is a random choice from the entire population proportional to fitness, requiring that all f_i have been computed to use as weights. This results in dependencies $\forall i \in N : (s_i, r)$. The selection of d is a random choice from $\Omega(r)$ (weighted only by edge weight), and thus depends only on r, yielding a dependency (r, d).

Next, $\forall i \in N : s_i'$ is set to s_r if $d = i$, or s_i otherwise. This means that s_i' could depend on r or s_i conditional on d, yielding dependencies $\forall i \in N : (s_i, s_i') \wedge (r, s_i') \wedge (d, (s_i, s_i')) \wedge (d, (r, s_i'))$. Finally, fitness of each agent f_i' must be recomputed if $i \in \Psi(d)$, in which case s_i' would depend on r or simply set to f_i otherwise.

In summary, a single round of **BD** is represented by the following CTDG:

$$
\begin{aligned}
T &= \{s_i \mid i \in N\} \cup \{s_i' \mid i \in N\} \cup \{d, r\} \\
D &= \{(s_i, r) \mid i \in N\} \cup \{(r, d)\} \\
&\quad \cup \{(s_i, s_i') \mid i \in N\} \cup \{(d, (s_i, s_i')) \mid i, \in N\} \\
&\quad \cup \{(r, s_i') \mid i \in N\} \cup \{(d, (r, s_i')) \mid i \in N\}
\end{aligned}
$$

Derivations of CTDGs for single rounds of **DB**, **PC**, **BR**, **BDD**, and **DBD** are similar, and are provided in Appendix A for brevity. However, all resulting CTDGs are depicted visually in Figures 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7, which should lend some intuition as to their utility.

Mutation may be added to any process with a straightforward modification of its TCDG. It involves the creation of a new task μ, which computes whether a mutation will occur and, if so, samples a type from the type space as the mutant. Any dependency $(t, s_i') \in D$ now depends on μ, since if a mutation does occur, s_i' no longer depends on anything but the mutant type. Indeed, if there is a mutation, (t, s_i') *only* depends on μ; thus, any $(t', (t, s_i')) \in D$ also depends on μ. In short, mutation can be represented by modifying $\langle T, D\rangle$ as follows, yielding $\langle T', D'\rangle$:

$$
\begin{aligned}
T' &= T \cup \{\mu\} \\
D' &= D \cup \{(\mu, (t, s_i')) \mid i \in N, (t, s_i') \in D\} \\
&\quad \cup \{(\mu, (t', (t, s_i'))) \mid i \in N, (t', (t, s_i')) \in D\}
\end{aligned}
$$

Once a CTDG has been determined, it can be algorithmically translated into a greedy intra-round parallel scheduler by repeated application of $\overset{R}{\Rightarrow}$ and $\overset{C}{\Rightarrow}$. Consider the CTDG of **BD**, in the context of the computational model. The computation begins in state $\langle \sigma_0, \emptyset, D\rangle$.

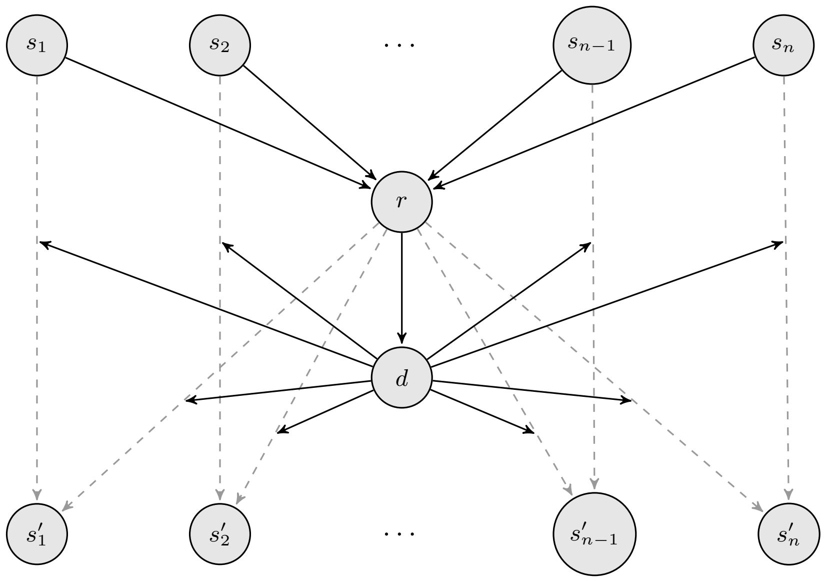


Figure 3.2: CTDG for a single round of **BD** without mutation.

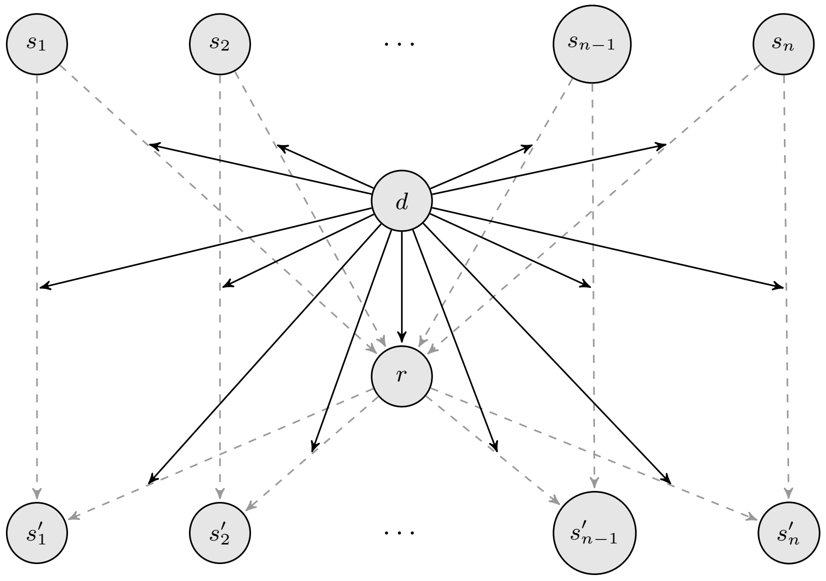


Figure 3.3: CTDG of a single round of **DB** without mutation.

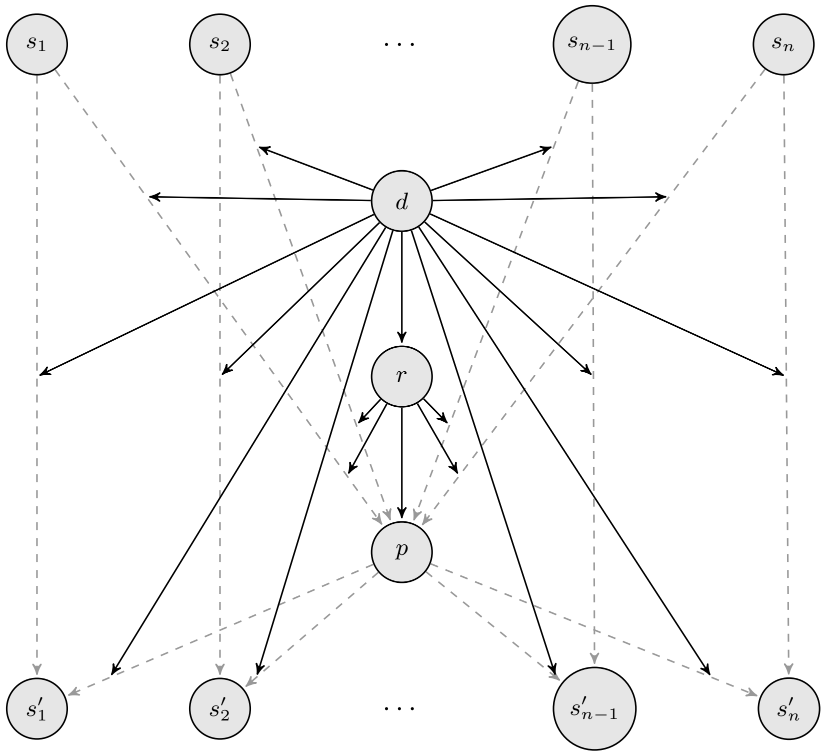


Figure 3.4: CTDG of a single round of **PC** without mutation.

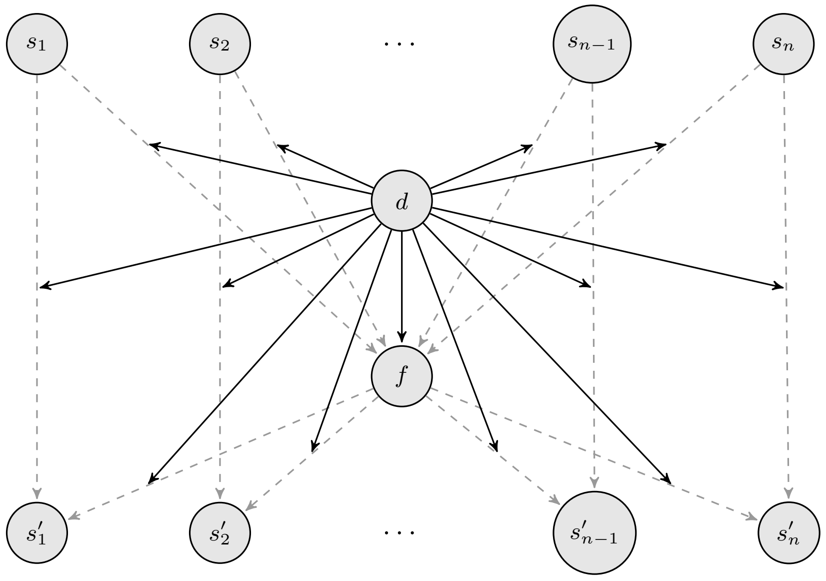


Figure 3.5: CTDG of a single round of **BR** without mutation.

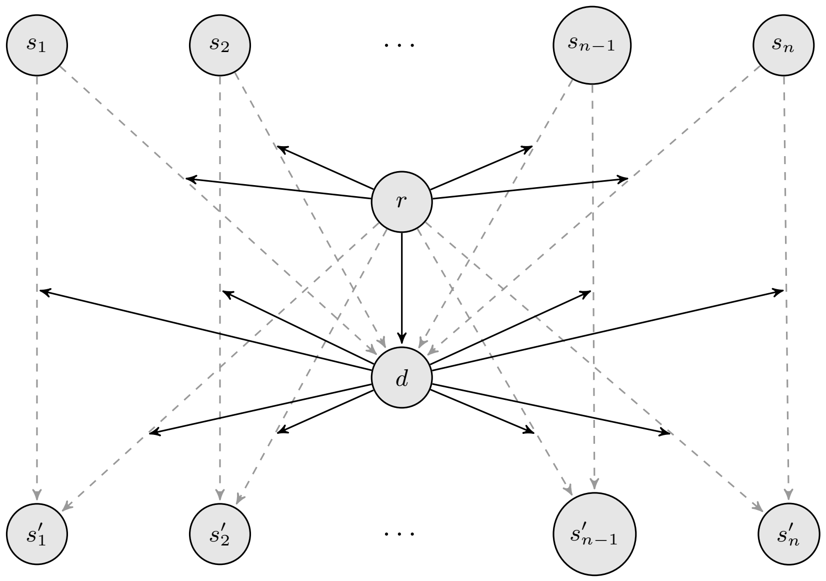


Figure 3.6: CTDG of a single round of **BDD** without mutation.

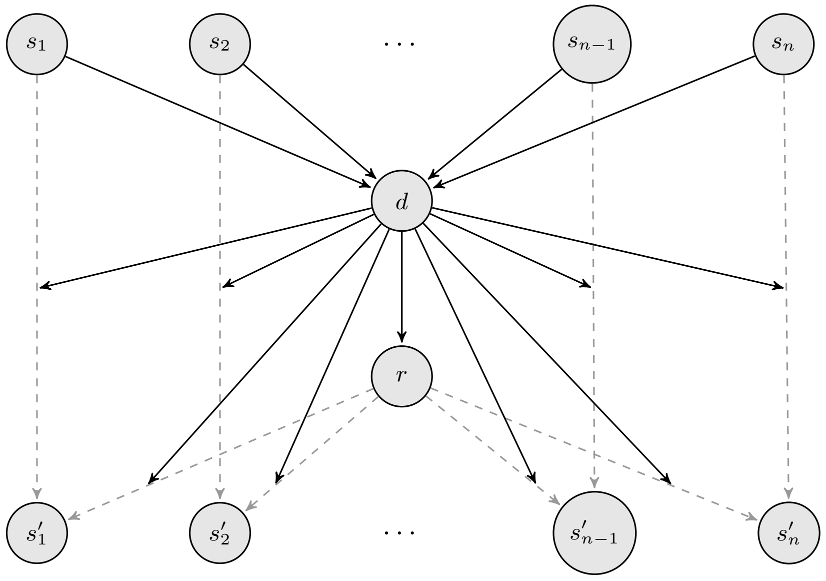


Figure 3.7: CTDG of a single round of **DBD** without mutation.

Informally, by inspection r is the only available task, and there are no resolvable edges. Once r has completed, d becomes the only available task, which can be approximated by sampling from the degree distribution[5]. Once it completes, tasks $\forall i \in N : s'_i$ become available.

Formally, there are three blocking computational groups (with applications of $\overset{R}{\Rightarrow}$ in between each line).

$$\langle \sigma_0, \emptyset, D \rangle \overset{C}{\Rightarrow} \langle \sigma_0[r \mapsto \pi_r(\sigma, R)], R, U \rangle$$
$$\langle \sigma, R, U \rangle \overset{C}{\Rightarrow} \langle \sigma[d \mapsto \pi_d(\sigma, R')], R', \emptyset \rangle$$
$$\langle \sigma, R, U \rangle \overset{C}{\Rightarrow} \langle \sigma[\forall i \in N : s'_i \mapsto \pi_{s'_i}(\sigma, R)], R, \emptyset \rangle$$

This yields the following scheduling procedure.

Algorithm 1 Update

```
function UPDATE($d, q$)
    if $q \neq s_d$ then
        for all $i \in \Psi(d)$ do
            $f_i \leftarrow f_i + w(i,d)\sigma(s_i, s_r) - w(i,d)\sigma(s_i, s_d)$
        $s_d \leftarrow q$
```

Algorithm 2 Round of **BD**

```
$r \leftarrow$ W-SELECT($N, f_x$)                    ▷ Weighted selection
$d \leftarrow$ W-SELECT($\Omega(r), w(r,x)$)
write $d \rightarrow r$
UPDATE($d, s_r$)
```

The work of such a scheduling procedure can be analyzed to determine theoretical speed-up. Note that since work is an asymptotic measure, it ignores constant factors. While theoretical values for such constants could be determined by reasoning about the precise computational steps involved in the task, in practice, they vary so widely across

[5]More precisely, this may be a slightly biased sample from the degree distribution, since the selection of an agent was not uniformly random. However, the effect of locality on fitness is assumed to be negligible over time.

hardware that they are not worth considering in theoretical analysis. Chapter 5 will investigate ways to measure and incorporate constants when designing an implementation for specific hardware.

Let $\overline{\Omega} = \frac{|E|}{|N|}$. When analyzing runtime, this will always be used as the value for $|\Omega(i)|$ and $|\Psi(i)|$ regardless of how i was selected. This is an approximation in two ways. First, it assumes a uniformly random selection of i, since any weighted or deterministic selection of i will generally introduce some bias. For example, if i is a random selection from the population weighted by fitness, it could be the case that average out-degree correlates with fitness. Such affects are assumed to be negligible, though it would be worth confirming this assumption in future work. Second, it always takes the mean degree instead of sampling from the degree distribution, which may also not be accurate in general. Again, this effect is assumed to be negligible across the large number of computations of this type being performed.

The basic justification for allowing these rough approximations is that, even if the effects of the assumptions turn out to be non-negligible, $|\Omega(i)|$, despite being a biased sample in general, can reasonably be assumed to strongly correlate with graph density, which is all that matters in the context of discussing scalability.

Considering the scheduler for **BD**, Update has asymptotic total work $T_1 = \overline{\Omega}$ and parallel work $T_K = \frac{\overline{\Omega}}{K}$.

BD therefore has sequential runtime

$$T_1 = |N| + \overline{\Omega} + \overline{\Omega}$$

and parallel runtime

$$T_K = \left\lceil \frac{|N| \log |N|}{K} \right\rceil + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil \log \overline{\Omega} + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil$$

The same procedure can be applied to the other processes, yielding the schedulers below.

Algorithm 3 Round of **DB**

```
$d \leftarrow \textsc{Select}(N)$                                ▷ Unweighted selection
$r \leftarrow \text{W-}\textsc{Select}(\Omega(r), f_x w(r,x))$
$\textsc{Update}(d, s_r)$
```

DB has sequential runtime

$$T_1 = 1 + \overline{\Omega} + \overline{\Omega}$$

and parallel runtime

$$T_K = 1 + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil \log \overline{\Omega} + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil$$

Algorithm 4 Round of **PC**

```
$d \leftarrow \textsc{Select}(N)$
$r \leftarrow \text{W-}\textsc{Select}(\Omega(r), w(r,x))$
$c \leftarrow \theta(f_d, f_r)$
if $c$ then
    $\textsc{Update}(d, s_r)$
```

PC has sequential runtime

$$T_1 = 1 + \overline{\Omega} + 1 + \overline{\Omega}$$

and parallel runtime

$$T_K = 2 + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil \log \overline{\Omega} + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil$$

BR has sequential runtime

$$T_1 = 1 + |\mathbf{Q}|\overline{\Omega} + \overline{\Omega}$$

and parallel runtime

$$T_K = 1 + \left\lceil \frac{|\mathbf{Q}|\overline{\Omega}}{K} \right\rceil + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil$$

Algorithm 5 Round of **BR**

```
function F$(i, q)$
    $f \leftarrow 0$
    for all $j \in \Omega(i)$ do
        $f \leftarrow f + w(i,j)\sigma(q, s_j)$
    return $f$
$d \leftarrow$ SELECT$(N)$
$q \leftarrow$ MAX-ARG$(\mathbf{Q}, F(d,x))$
UPDATE$(d, q)$
```

Algorithm 6 Round of **BDD**

```
$r \leftarrow$ SELECT$(N)$
$d \leftarrow$ W-SELECT$(\Omega(r), \frac{w(r,x)}{f_x})$
UPDATE$(d, r)$
```

BDD has sequential runtime

$$T_1 = 1 + \overline{\Omega} + \overline{\Omega}$$

and parallel runtime

$$T_K = 1 + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil \log \overline{\Omega} + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil$$

Algorithm 7 Round of **DBD**

```
$d \leftarrow$ W-SELECT$(N, \frac{1}{f_x})$
$r \leftarrow$ W-SELECT$(\Omega(d), w(d,x))$
UPDATE$(d, r)$
```

DBD has sequential runtime

$$T_1 = |N| + \overline{\Omega} + \overline{\Omega}$$

and parallel runtime

$$T_K = \left\lceil \frac{|N| \log |N|}{K} \right\rceil + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil \log \overline{\Omega} + \left\lceil \frac{\overline{\Omega}}{K} \right\rceil$$

3.4 Inter-round Parallelization

Another approach is to attempt to compute multiple rounds in parallel, in addition to parallelizing the computations within each round. For such inter-round parallelizability to be possible, at least some tasks must be available in a round without having computed all tasks in the previous round.

In the context of single-round CTDGs, this means that the input tasks of each round can no longer be assumed to be computed (or computable). This can be studied by considering the following variant of a single-round CTDG $\langle I \cup O \cup V, D\rangle$: $\langle O \cup V, D\rangle$. This is a sort of hack of notation, since D still contains dependencies on $t \in I$, so anything that depends on such a t cannot be computed.

An initial question to ask is whether it is possible to do any work on a round without being able to compute $t \in I$. This leads to the following definition.

Definition 3.4.1 (Weak Inter-round Parallelizability)**.** A process with single-round CTDG $\langle I \cup O \cup V, D\rangle$ is *weakly inter-round parallelizable* if, for CTDG $\langle O \cup V, D\rangle$, $\exists \sigma', U, R : \langle \sigma_0, D, \emptyset\rangle \Rightarrow \langle \sigma', U, R\rangle \wedge \exists t \in T : t \in \sigma'$.

Weak inter-round parallelizability is a minimal requirement: it simply means that at least some tasks in a round can be computed in the absence of input. Note that any weak inter-round parallelizability is embarrassingly parallel across rounds by definition. Also, note in general that mutation introduces weak inter-round parallelizability in any process, since μ never has any dependencies. Inspection of the CTDGs for single rounds of the processes yields the following theorem.

Theorem 3.4.1 (Weak Inter-round Parallelism)**.** *Without mutation, processes **DB**, **PC**, **BR**, and **BDD** are weakly inter-round parallelizable. Processes **BD** and **DBD** are not.*

Proof. Proofs follow directly the CTDGs given in Appendix A. □

Round	I^+	O^+
1	$\{3, 4\}$	$\{5\}$
2	$\{3, 4\}$	$\{4, 1\}$
3	$\{5, 6\}$	$\{1\}$
4	$\{1, 8, 10\}$	$\{8, 9\}$
...	...	...

Table 3.1: Example round dependency table, with $N = \{1, \ldots, 10\}$.

This is promising: many of the processes have at least some useful work that is embarassingly parallel across rounds. Unfortunately, by inspection of the CTDGs, the only tasks being computed are the random uniform selection of d or r (depending on the process), leaving most of the round unfinished, and seemingly unable to progress.

However, the computation of these tasks also allows some dependencies to be resolved. Consider **DB**, where the computation of d in fact allows *all* dependencies to be resolved. In particular, it allows dependencies (s_i, t) to be resolved – that is, all dependencies on inputs. Thus, I can be partitioned into two sets: $I^- = \{i \in N \mid \forall t \in T \setminus \{s'_i\} : \nexists (s_i, t) \in R\}$ and $I^+ = \{i \in N \mid \forall t \in T \setminus \{s'_i\} : \exists (s_i, t) \in R\}$.

Note that, by definition, $\forall i \in I^-$ no $t \in T$ other than s'_i can depend on s_i. Thus, no tasks in the round depend on i. Conversely, there *must* be tasks in the round which depend on $i \in I^+$. This is extremely useful: it means that despite the lack of computed inputs, it is possible to determine exactly which inputs the round depends on. Definitions for O^+ and O^- follow similarly, since all (t, s'_i) are also resolvable, such that it is also possible to determine which outputs the round will affect. Note that it is possible to do this for any round, in any order, regardless of which rounds have been computed, producing something like Table 3.1.

Formalizing this idea across processes produces the following definitions and theorem.

Definition 3.4.2 (Input-Resolvability)**.** A CTDG $\langle I \cup O \cup V, D\rangle$ is *input-resolvable* if, for CTDG $\langle O \cup V, D\rangle$, $\exists \sigma, U, R : \langle \sigma_0, D, \emptyset\rangle \Rightarrow \langle \sigma, U, R\rangle \wedge \forall i \in I : \nexists (i, t) \in U$. If a CTDG is input-resolvable, then I^+ can be computed.

Definition 3.4.3 (Output-Resolvability)**.** A CTDG is *output-resolvable* if $\exists \sigma, C', D'$: $\langle V \cup O, \sigma_0, \emptyset, C, D\rangle \rightarrow^* \langle V \cup O, \sigma, C', D'\rangle \wedge \forall o \in I, \forall (t, o) \in C : (t, o) \notin C'$. If a CTDG is output-resolvable, then O^+ can be computed.

Definition 3.4.4 (Strong Resolvability)**.** A CTDG is *strongly resolvable* if it is both input- and output-resolvable.

Theorem 3.4.2 (Resolvability of Processes)**.** ***DB**, **PC**, and **BR** are strongly resolvable, **BDD** is input-resolvable but not output-resolvable, and **BD** and **DBD** are neither input- nor output-resolvable.*

Proof. See Appendix A. □

The implications of strong resolvability are very powerful. Consider again Table 3.1, and note that round i depends on round j iff $I_i^+ \cap O_j^+ \neq \emptyset$. Thus, such a table can be translated to a non-conditional dependency graph which describes which rounds depend on which other rounds. For instance, in Table 3.1, round 3 depends on round 1 (because the former affects 5 and the latter depends on it), and round 4 depends on rounds 2 and 3. If two rounds are simultaneously available (i.e. all of the rounds they depend on have been computed), they can be computed in parallel. The exact details of this process are implemented in Algorithm 8, which can be used to schedule the computation of entire rounds.

The general idea of the algorithm is to keep the size of the dependency graph as small as possible (noting that nodes can be deleted once their rounds have been computed), since it does have to fit in memory, as do the partial computations done on each round in it. This is done by only adding dependencies to the graph when there are available processors but no available rounds.

To briefly summarize the main result: processes are strongly resolvable if, for any round, the sets of inputs and outputs which the round depends on and affects, respec-

Algorithm 8 Inter-round scheduler for x rounds

```
Require: empty queue $\gamma$                                  ▷ Available rounds

  function ADD($i, D, A$)
      $\delta_i \leftarrow \emptyset$
      $\rho_i \leftarrow \emptyset$
      for all $d \in D$ do
          for all $j \in AFF_d$ do
              $\rho_j \leftarrow \rho_j \cup \{i\}$
              $\delta_i \leftarrow \delta_i \cup \{j\}$
      for all $a \in A$ do
          $AFF_a \leftarrow AFF_a \cup i$

  function UPDATE($i$)
      for all $j \in \rho_i$ do
          $\delta_j \leftarrow \delta_j \setminus \{i\}$
          if $\delta_j = \emptyset$ then
              del $\delta_j$
              push $j$ to $\gamma$
      del $\rho_i$

  procedure SCHEDULE($x$)
      $i \leftarrow 0$
      when processors available do
          if $\gamma$ not empty then
              pop $i$ from $\gamma$
              async
                  COMPUTE($i$)
                  UPDATE($i$)
          else if $i < x$ then
              $i \leftarrow i + 1$
              async
                  $(D, A) \leftarrow$ RESOLVE($i$)
                  ADD($i, D, A$)
```

tively, can be computed without having computed the inputs. Such processes can potentially benefit from having rounds computed in parallel, since the resulting relationships between rounds can be arranged in a dependency graph where multiple rounds may be available concurrently.

To consider the resulting theoretical speed-up of such an approach, first note that when a set of rounds *can* be computed in parallel using this method, doing so is in fact embarrassingly parallelizable. Thus, insofar as the method works, it is extremely efficient, especially since the discrete task of computing a round is very large.

To determine how frequently rounds can actually be computed in parallel, it is useful to consider the average size of I^+ and O^+. Using the $\overline{\Omega}$ approximation established earlier yields the following values for each strongly resolvable process[6]. Let $d^+ = \rho(|N-1|)+1$, the expected number of deaths and replacements in each round.

Process	I^+	O^+
DB	$\overline{\Omega}d^+$	$\overline{\Omega}d^+$
PC	$2\overline{\Omega}d^+$	1
BR	$\overline{\Omega}d^+$	1

This immediately provides the intuitive result that the benefit of inter-round parallelizability is inversely proportional to population density, since greater numbers of dependencies increase the chance that a given round will depend on another.

More specifically: suppose that k rounds are either currently being computed or have been resolved and are in the queue. What is the chance that a newly resolved round does not depend on any of these k rounds? For **DB**, **PC**, and **BR**, assuming that the dependencies for each round are sampled uniformly randomly from the population without replacement (with replacement between rounds), then the probability that none of a given round's dependencies depend on any of the queued round's dependencies is

[6]Note that this is a very rough approximation, since it assumes a random graph.

approximately

$$p_{\mathbf{DB}} = \left(\prod_{i=0}^{\overline{\Omega}d^+} \frac{|N| - i - \overline{\Omega}d^+}{|N| - i} \right)^k$$

$$p_{\mathbf{PC}} = \left(\frac{|N| - 2\overline{\Omega}d^+}{|N|} \right)^k$$

$$p_{\mathbf{BR}} = \left(\frac{|N| - \overline{\Omega}d^+}{|N|} \right)^k$$

For $|N| \gg \overline{\Omega}$, small k, and small ρ, this probability is reasonably high, particularly for **BR** and **PC**. Specifically, inter-round parallelization should perform very well on sparse populations when $\rho \approx 0$. However, $\rho \rangle 0$ and $\overline{\Omega} \gg 1$ negatively compound, suggesting that inter-round parallelization will perform poorly on denser graphs, or on processes with a nontrivial synchronization rate.

Chapter 4

Approximation

4.1 Motivation

So far, I have analyzed parallel models of processes which produce exactly equivalent simulations as the sequential models on which they are based. In this chapter, I investigate models which relax this constraint, allowing for parallel approaches yielding similar, but not exactly equivalent, dynamics.

The motivation for doing so is, informally, the following. Computing rounds in parallel, *when it is possible to do so*, has excellent speed-up and scalability properties, due to its embarassingly parallel nature and because the tasks being assigned to individual processors are very large, minimizing parallel overhead. Indeed, if each processor could always be responsible for computing an entire round, this would achieve $E_K \approx 1$, regardless of population size or structure. Thus, it is worth attempting to increase the frequency with which inter-round parallelization can be employed without resorting to intra-round approaches, which are generally significantly less efficient. Depending on the specific evolutionary model, it may even be worth sacrificing some accuracy of the simulation to achieve this.

If accuracy is to be sacrificed, it should be done in a way that maximizes the resulting gains in efficiency. Ideally, any error that is introduced should also be, informally, unbiased, such that it can be interpreted as something like noise, as opposed to as a force which pushes the results of the simulation in a particular direction.

4.2 Dependency Dropping

An approximation method is proposed which is a variant of the inter-round parallel scheduler introduced in Chapter 3 as Algorithm 8.

The method makes use of *dependency dropping*, where rounds will sometimes be considered available for computation even when all of their input dependencies have not been computed. The rate at which this happens is inversely correlated to the fraction of "missing" dependencies, strongly favoring rounds which have most or all of their dependencies satisfied in order to minimize error. If a round with missing dependencies is computed, it simply uses the most recent data available for those inputs.

One approach to dependency dropping is given in Algorithm 9. There are two main differences between Algorithms 8 and 9. The first is the introduction of a new variable α_i for each resolved round i, which simply records the total number of input-dependencies of i and is destroyed when the round is computed. The second is a change in the check of whether to make a round available when one of its dependencies is resolved. Specifically, instead of simply checking whether the set of remaining dependencies is empty, this decision is outsourced to $\textsc{Drop} : \mathbb{N} \times \mathbb{N} \to \textbf{bool}$, which takes α_i and $|\delta_i|$ (the number of remaining unresolved dependencies) as arguments.

The only formal restriction on $\textsc{Drop}$ is that its probability of returning **true** must increase monotonically in $1 - \frac{|\delta_i|}{\alpha_i}$, and must always return **true** when $|\delta_i| = 0$. That is, a round should always be made available if it has no remaining dependencies, and it would not make sense for a round missing a greater fraction of its dependencies to be made available more often than if it were missing fewer dependencies.

A reasonable implementation is $\textsc{Drop}(\alpha, \delta) = U < e^{-\frac{\delta}{\tau\alpha}}$ where $U \sim \mathrm{Unif}(0, 1)$ for

Algorithm 9 Inter-round scheduler for x rounds using dependency dropping

```
Require: empty queue $\gamma$                                   ▷ Available rounds

  function Add($i, D, A$)
      $\delta_i \leftarrow \emptyset$
      $\rho_i \leftarrow \emptyset$
      $\alpha_i \leftarrow |D|$                              ▷ Total number of round dependencies
      for all $d \in D$ do
          for all $j \in AFF_d$ do
              $\rho_j \leftarrow \rho_j \cup \{i\}$
              $\delta_i \leftarrow \delta_i \cup \{j\}$
      for all $a \in A$ do
          $AFF_a \leftarrow AFF_a \cup i$

  function Update($i$)
      for all $j \in \rho_i$ do
          $\delta_j \leftarrow \delta_j \setminus \{i\}$
          $drop \leftarrow$ Drop($|\delta_j|, \alpha_j$)          ▷ Dependency dropping decision
          if $drop$ then
              del $\delta_j, \alpha_j$
              push $j$ to $\gamma$
      del $\rho_i$

  procedure Schedule($x$)
      $i \leftarrow 0$
      when processors available do
          if $\gamma$ not empty then
              pop $i$ from $\gamma$
              async
                  Compute($i$)
                  Update($i$)
          else if $i < x$ then
              $i \leftarrow i + 1$
              async
                  $(D, A) \leftarrow$ Resolve($i$)
                  Add($i, D, A$)
```

some $\tau \in \mathbb{R}^+$, which is a sort of "optimism" parameter. $\tau \to 0$ represents the limit where a round is only made available if $|\delta_i| = 0$, and $\tau \to \infty$ represents the limit where rounds are always made available, regardless of dependencies.

Chapter 5

Implementation

5.1 Data Structures

I now examine how this theoretical model can be implemented in practice. Specifically, I determine the optimal data structure representations of various portions of the model, discuss their respective memory usage, and briefly discuss some optimizations of the implementations of parallel algorithms.

Let M represent the number of bits of memory in the system. Let b_S be the number of bits required to store the index of a member of set S, $\lceil \log_2 |S| \rceil$.

Population structure

Of the many standard representations of weighted digraphs, the chosen representation should have a few desirable properties: fast lookup of $\Omega(i)$ and $\Psi(i)$ for arbitrary $i \in N$, and minimal memory usage.

One option is an adjacency matrix, which requires $|N|^2$ bits to represent an unweighted graph and $32|N|^2$ bits to store a graph with 32-bit (integer or floating point) edge weights. Lookup of both $\Omega(i)$ and $\Psi(i)$ is $O(|N|)$. The generally superior option is to use two adjacency lists to store $\Omega(i)$ and $\Psi(i)$ for all $i \in N$, respectively. This requires approximately $4b_N|E|$ bits to represent an weighted graph, and $(4b_N + 32)|E|$ to represent a graph with 32-bit edge weights, and provides $O(1)$ lookup for both $\Omega(i)$ and $\Psi(i)$. Thus, it has strictly better lookup time, and better memory usage for reasonably sparse population structures (specifically, when density $< \frac{1}{4b_N}$, which is typical). An adjacency matrix should only be considered for extremely large, dense graphs for which the simulation would otherwise not fit into memory.

Type space and population state

b_Q bits are required to represent a single type in a finite type space (by index). Infinite type spaces are technically impossible to implement, but can be approximated using floating point values, which is equivalent to discretizing them by approximately equal intervals into a finite space of length $|Q|$.

The entire type space does not need to be stored in memory. However, in sequential and intra-round parallelized implementations, the current state of the population does, the size of which depends on the size of the type space, since for $|N|$ agents, $b_N|N|$ bits are required to store the state of a single round.

Historical state also needs to be preserved in memory until it can be flushed to disk as output. If the disk write speed is too slow to keep up with the speed of the simulations, then memory use grows infinitely over time, and the simulation may need to regularly pause to allow the state buffer to be written to disk and cleared. However, there is a significant optimization to be made by using round deltas (i.e. records of only the agents whose state has changed between rounds), which reduces the representation size of the state of any round beyond the first by an expected factor of p (the synchronization rate). This hugely decreases both memory usage and data written to disk[1]. Thus, in practice, disk write speed is never a bottleneck.

[1]Note that this has the trade-off of requiring more processing when the resulting output is read as input, since the full state of each round must be reconstructed.

Fitness

Fitness is directly computable from population state. However, this computation is relatively expensive, and should only be done when needed. In particular, fitness for all agents can be stored in its own array, and only computed in its entirety once at the beginning of the simulation. Small updates of this array suffice otherwise. In particular, whenever the state of agent i changes, the fitnesses of agents in $\Psi(i)$ must be updated. They need not be entirely recomputed; rather, the following optimization can be used: $\forall j \in \Psi(i) : f'_j = f_j + w(j,i)(\sigma(s'_j, s'_i) - \sigma(s'_j, s_i))$. This simply subtracts the old portion of j's fitness caused by i, and adds the new portion.

Interaction function

For finite, relatively small type spaces (such as the one used in experiments in Chapter 6), all $|Q|^2$ possible values of the interaction function are typically pre-computed, such that an interaction computation is simply a unit-time lookup from a $|Q| \times |Q|$ array. This only needs to be done once across all simulations.

Another approach for larger type spaces is to allocate an array of the same size, but lazily compute and cache interaction values as needed (across all simulations) rather than pre-computing them. This is worthwhile if a $|Q|$ is small enough that a lookup table fits into memory, but large enough that, given the number of simulations and the number of rounds in each one, in expectation, at least some values of the table will never need to be computed.

When the type space is so large that an interaction lookup table cannot fit into memory, interactions must be computed on-the-fly. In this case, only the interaction function is stored, requiring a constant, trivially small amount of memory.

5.2 Algorithms

Sequential

The sequential algorithm does not require a scheduler, since it does not matter in which order tasks are computed. Thus, the sequential algorithm has essentially no overhead, and should therefore scale at a rate very close to the T_1 values calculated in Chapter 3.

Intra-round Parallel

One consideration in intra-round parallelization is whether or not to parallelize computations (or random weighted selections) across $\Omega(i)$. In particular, for sparse population structures where $\overline{\Omega}$ is small, and for cases where the task to be performed for each neighbor is small, the overhead of parallelizing this task generally dominates the computation itself, such that parallelization actually results in a slow-down. It is only for comparatively dense graphs where $\overline{\Omega}$ approaches $|N|$ that the overhead becomes worthwhile, such that parallelization yields any speed-up at all.

Fortunately, a reasonable cutoff threshold for $\overline{\Omega}$, only beyond which parallelization across $\Omega(i)$ is enabled, can be easily determined empirically for a specific implementation using a binary search, where the runtimes of sequential and parallel approaches are compared for candidate cutoff for $\overline{\Omega}$.

Inter-round Parallel

When inter-round parallelism is enabled, it is again worth considering whether to parallelize across $\overline{\Omega}$. For values of $\Omega(i)$ where parallism yields only marginal speed-up, it may not be worth enabling if there are other available rounds to compute but all processors are busy, because the former results in much greater efficiency.

Similarly, parallelizing weighted random selection in general should be questioned, even across $|N|$, since it yields strictly sub-linear speed-up. If the population is sufficiently sparse and/or there is sufficient dependency dropping for processors to consistently be occupied with inter-round parallelization, then parallel weighted random selection should be disabled entirely. Again, the proper cutoff rates can be determined empirically.

Chapter 6

Results

6.1 Parameter Space

I now use the implementation discussed in Chapter 5 to empirically test the results of Chapters 3 and 4. Specifically, speed-up is tested across parallelization methods, processes, synchronization rates, population structures, and approximation rates. For a summary of these varied parameters, see Table 6.1. All experiments were performed on an 8-CPU computer with 64GB of memory.

There are even more parameters that could be varied, but have instead been fixed across all experiments in this chapter. (See Table 6.2 for a complete list.) The decision to fix each of these variables is justified below, based on their lack of influence on, or

Parameter	Domain	Varied
Population structure	Arbitrary weighted digraph	Various
Process	Arbitrary update rule	**BD**, **DB**, etc.
Synchronization rate	$[0, 1]$	$[0, 0.5, 1]$
Parallelization	Sequential, Intra, Inter	All
Dependency drop rate	$[0, 1]$	$[0 : 1 : 0.05]$

Table 6.1: Parameters varied across the experiments.

Parameter	Domain	Fixed at
Type space $\mathbf{Q}$	Arbitrary set	$\{C, D\}$
Initial population composition s_0	$s_0 \in Q^{\vert N\vert}$	Random sample
Interaction function σ	$Q \times Q \to \mathbb{R}_0^+$	PD
Number of rounds	$\mathbb{N}^+$	1,000,000
Number of processors	$\mathbb{N}^+$	8
Mutation rate	$[0, 1]$	0.02

Table 6.2: Parameters fixed across all experiments.

predictability with respect to, computational speed and memory usage.

Type space

Size of the type space generally does not influence computation time, since the type space is never iterated over during a simulation, except in the case of **BR**[1].

Interaction function

For a pre-computed interaction lookup table, the runtime of this pre-computation is influenced by the complexity of the interaction function. However, for type spaces small enough for a pre-computed lookup table, this runtime is generally trivial compared to that of the simulations themselves. Further, the process of pre-computation is embarassingly parallelizable across the type space.

For on-the-fly interaction computation, complexity of the interaction function does have a noticeable effect on computation time, since it must be called $|\Omega(i)|$ times every time an agent $i \in N$ needs its fitness recomputed. However, while this has not been confirmed empirically, I would expect the runtime increase to uninterestingly scale almost linearly in K when population density dominates the number of cores, since a single round of fitness computations is embarassingly parallelizable.

[1]While outside the scope of this thesis, it would be relatively straightforward to analyze the effect of type space size on **BR** using the machinery developed in Chapter 3.

Initial population composition

Initial population composition generally has little to no effect on computation time, except in very contrived cases, such as in an initially homogenous population, where the optimization described in Chapter 5 allows state/fitness updates to be skipped if the state does not change between two rounds (i.e. if the reproducing agent has the same type as the dying agent).

Number of rounds/simulations

Number of rounds does, of course, influence computation time. For sequential and intra-round parallelized implementations, it is reasonable to expect an almost exactly linear increase. For inter-round parallelized implementations, the increase is very close to linear after sufficiently many rounds. (The last few rounds act as "bottlenecks", but this effect is negligible when the number of rounds is large.) For similar reasons, the number of simulations should have an almost exactly linear effect on computation time.

Mutation rate

Mutation rate influences computation time in complex ways, but its analysis is straightforward given the machinery introduced in Chapters 3 and 4. However, the specific study of its effects is beyond the scope of this thesis. Further, it is rarely varied in experiments; it is typically set to a constant low value as a mechanism to introduce new strategies into the population for non-innovative processes.

6.2 Speed-up

Speed-up is tested across parallelization methods, processes, synchronization rates, and population structures. Specifically, experiments are run across all processes described in Chapter 2, ρ at 0, 0.5, and 1 (for processes which support synchronization), and two population structures, one dense and one sparse. Both populations were generated by a copying model, with $|N| = 10000$. On the dense population, $|E| = 1185208$, and on the sparse one, $|E| = 108362$.

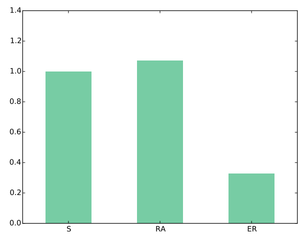
The results for the sparse population are shown in Figures 6.1 and 6.2, and those for the dense population in Figures 6.2 and 6.4.

Looking first across synchronization rate on the sparse graph, note that intra-round speed-up consistently improves with respect to ρ. Intuitively, this is because a higher synchronization rate results in more tasks that can be meaningfully parallelized within rounds. Intra-round parallelization generally performs terribly when $\rho = 0$, with runtime comparable to that of the sequential algorithm. This is due to the overhead of parallelism for smaller tasks outweighing the benefit.

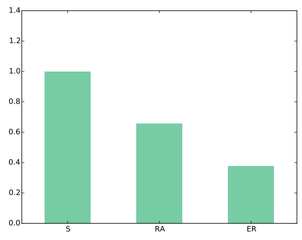
Inter-round parallization performance stays roughly constant, or gets worse, as ρ increases. This is explained by the fact that increasing ρ increases the average size of a round's dependency set. In fact, if not for the gains caused by its incorporation of intra-round parallism, inter-round parallelization would likely perform strictly worse than the sequential algorithm for large ρ. One somewhat strange result is that inter-round parallelism performs slightly better than intra-round parallelism for **PC** with $\rho = 1$; presumably this is just noise, since it does not make sense as a general result.

Looking next across processes on sparse graph, the following trends are observed. Inter-round parallelization performs poorly on **BD**, **BDD**, and **DBD**, which is expected, since these processes are not inter-round parallelizable. Thus, due to the overhead of keeping track of dependencies, it performs consistently slightly worse than intra-round parallelization for these processes. Intra-round parallelization performs particularly well on **BD** and **DBD**, since they both involve parallelization across N.

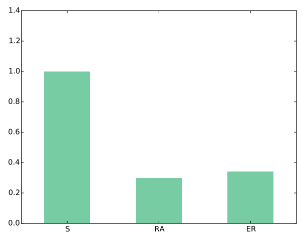
Processes **DB**, **PC**, and **BR** display similar speed-ups for both intra- and inter-round parallelization. **BR** yields slight improvements for intra-round parallization, likely due to its lack of weighted random selection, which is not as efficient to parallelize.



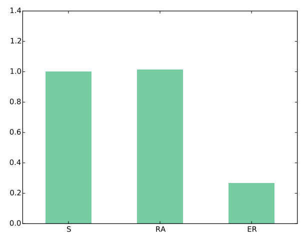
(a) DB with $\rho = 0$



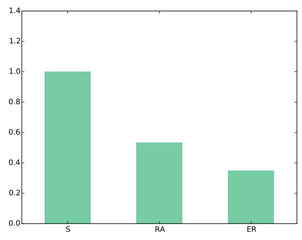
(b) DB with $\rho = 0.5$



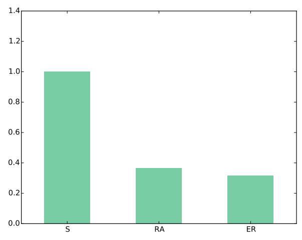
(c) DB with $\rho = 1$



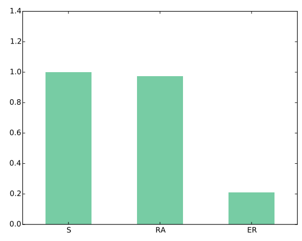
(d) PC with $\rho = 0$



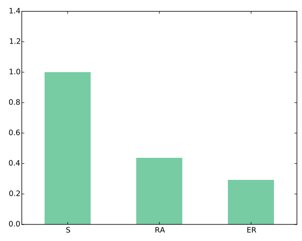
(e) PC with $\rho = 0.5$



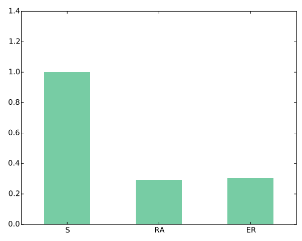
(f) PC with $\rho = 1$



(g) BR with $\rho = 0$

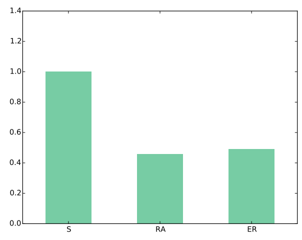


(h) BR with $\rho = 0.5$

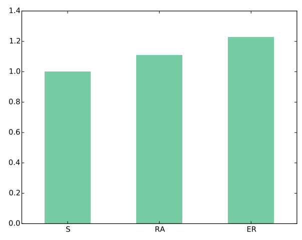


(i) BR with $\rho = 1$

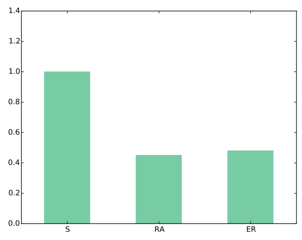
Figure 6.1: Relative mean runtimes across various synchronizable processes (from top to bottom) and synchronization rates (from left to right) on a sparse graph. Each chart shows data for sequential, intra-round parallel, and inter-round parallel algorithms, from left to right.



(a) BD

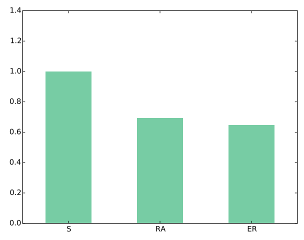


(b) BDD

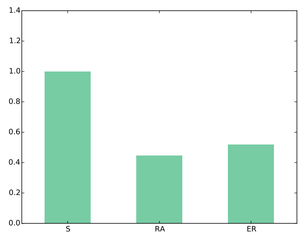


(c) DBD

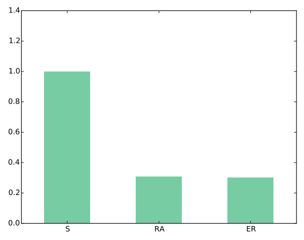
Figure 6.2: Relative mean runtimes across various non-synchronizable processes, using the same population structure as above.



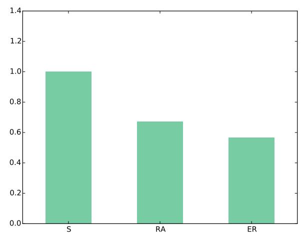
(a) DB with $\rho = 0$



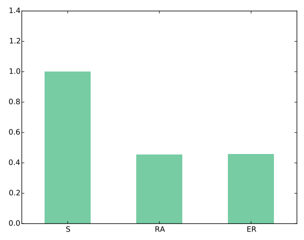
(b) DB with $\rho = 0.5$



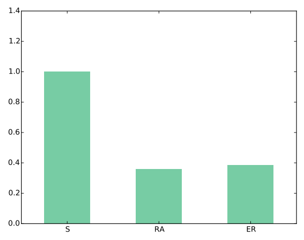
(c) DB with $\rho = 1$



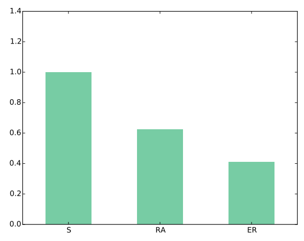
(d) PC with $\rho = 0$



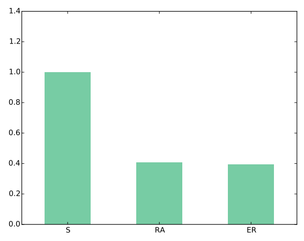
(e) PC with $\rho = 0.5$



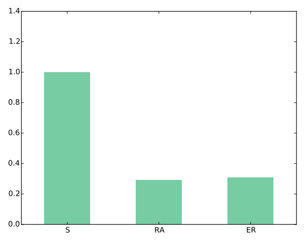
(f) PC with $\rho = 1$



(g) BR with $\rho = 0$

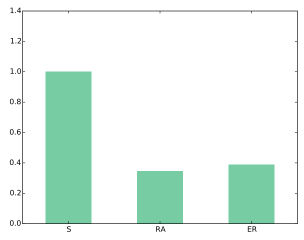


(h) BR with $\rho = 0.5$

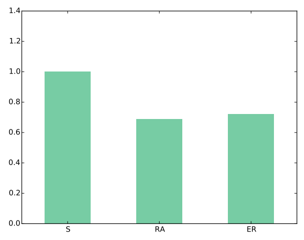


(i) BR with $\rho = 1$

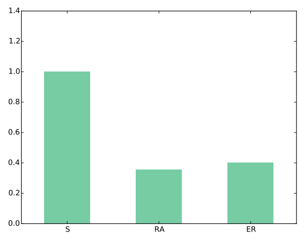
Figure 6.3: Relative mean runtimes across various synchronizable processes (from top to bottom) and synchronization rates (from left to right) on a dense graph. Each chart shows data for sequential, intra-round parallel, and inter-round parallel algorithms, from left to right.



(a) BD



(b) BDD



(c) DBD

Figure 6.4: Relative mean runtimes across various non-synchronizable processes, using the same population structure as above.

Finally, looking between the sparse and dense graphs, as expected, intra-round parallelism generally performs much better on the dense graph, whereas inter-round parallelism suffers. In general, the two methods perform at more comparable rates. This is simply because intra-round parallelism is yielding most of the speed-up across all processes on these graphs.

In general, there is a very close match between the theoretical predictions of Chapters 3 and 4 and these results.

Chapter 7

Conclusion

7.1 Summary

In Chapters 3 and 4, I introduced a general model for computability of the stochastic evolutionary processes described in Chapter 2. I then proposed intra- and inter-round parallelism methods, including an approximation variant, and analyzed their theoretical speed-up, finding both approaches to be highly scalable, with varying asymptotic efficiency dependent on population size, density, process, and synchronization rate.

Implementation considerations discussed in Chapter 5 yielded a more practical view of the effects of these parameters. In particular, both kinds of parallelism benefit from increased population size, and intra-round parallelization is optimized by denser population structures and processes which involve frequent iteration over $i \in N$ or high synchronization rates, whereas inter-round parallelization performs optimally on sparse population structures, and on processes which tend to have small round dependency sets relative to population size. In Chapter 6, these theoretical results were confirmed empirically.

7.2 Future Work

I now conclude by suggesting future work, including applications of these parallel methods to open problems in evolutionary dynamics and other domains. I also discuss potential extensions and improvements of these methods.

Improving theoretical bounds

Currently, the theoretical analysis of speed-up for parallelization methods uses a number of approximations. Specifically,

- It makes fairly liberal assumptions about memory access, such as by assuming a PRAM CREW system.
- It does not consider algorithmic constants. For example, if an algorithm requires iterating over input of size n twice, its work is still assumed to be n.
- It ignores some aspects of parallel overhead, such as synchronization and messaging between processors.

In the context of scalability, which is where theoretical speed-up is predominantly discussed, the error introduced by such approximations is generally not relevant. Further, these approximations, of course, do not affect the actual speed-up of the models, and the empirical data supports the theoretical results, suggesting that the approximations are reasonable.

However, in order to better understand the relative performance of parallel methods across the parameter space without having to rely on empirical methods for verification, it would be worth attempting to reduce the use of such approximations, or to more strictly bound them when this is not possible. Specifically, it may be possible to reason more concretely about the amount of work involved in various tasks on particular hardware, allowing constant factors to be determined and used in efficiency calculations.

Other approximation approaches

Given the positive results obtained by the approximation approach described in Chapter 4, it is worth considering how else the use of slight approximations could provide further speed-up. For example, different implementations of Drop could be explored, as well as different locations where the dependency dropping decision is made. Further, beyond dependency dropping, perhaps there are other reasonable approximations that could be introduced in other parts of the simulations.

Applications

In their current state, the parallelization methods can be readily applied to a variety of open problems in various domains. Examples include:

Emergence of cooperation As mentioned briefly in Chapter 2, there are many open questions surrounding the effect of population structure on the evolution of the emergence and persistence of cooperation, especially in the context of large-scale social networks [1].

Games with larger type spaces Traditionally, both analytical approaches and simulations have been limited in their ability to handle larger type spaces, partially because of the additional memory required to store them, but largely because of the increased number of rounds needed to simulate to ensure statistically valid results. As a specific example, there has been some work in representing strategies in Iterated Prisoner's Dilemma games as finite state automata [17], but not on structured populations.

Confirming assumptions Much work in applied evolutionary dynamics makes use of deterministic formulations of models, assuming that stochastic effects are negligible, since the former are generally much faster to simulate. Employing the speed-up gained by parallel approaches, it would be worth confirming that such assumptions are true, since stochastic effects are occasionally quite important [8].

Political influence A somewhat analogous problem to the evolution of cooperation in political science concerns how political beliefs emerge and spread, and the role that social networks play in influencing and catalyzing this process. Such problems can be studied using stochastic evolutionary models. However, this presents a formidable computational problem due the size of the populations involved, which parallelism can help reduce.

Cultural and linguistic evolution Games on graphs are well-suited to the study of both cultural and linguistic evolution, where cultural beliefs and linguistic tendencies flow along social networks. This type of research similarly suffers from computational restrictions due to the size of real-world populations.

Further applications are left as an exercise for the reader.

Appendix A

CTDG Derivations

DB

DB requires selections of an agent to reproduce and an agent to die, so $V = \{d, r\}$. The selection of d is a uniformly random choice from the entire population, so it has no dependencies. r is a weighted random choice from $\Omega(d)$ proportional to fitness and edge weight, so it depends on $\forall i \in \Omega(d) : s_i$. This yields dependencies $(d, r) \in D$ and $\forall i \in N : (s_i, r) \in D \wedge (d, (s_i, r)) \in D$.

Finally, $\forall i \in N : s'_i$ is set to s_r if $d = i$ or s_i otherwise, yielding dependencies $\forall i \in N : (s_i, s'_i) \wedge (r, s'_i) \wedge (d, (s_i, s'_i)) \wedge (d, (r, s'_i))$.

Thus, a single round of **DB** is represented by the following CTDG:

$$\begin{aligned} T &= \{s_i \mid i \in N\} \cup \{s'_i \mid i \in N\} \cup \{d, r\} \\ D &= \{(d, r)\} \cup \{(s_i, r) \mid i \in N\} \cup \{(d, (s_i, r)) \mid i \in N\} \\ &\quad \cup \{(s_i, s'_i) \mid i \in N\} \cup \{(d, (s_i, s'_i)) \mid i, \in N\} \\ &\quad \cup \{(r, s'_i) \mid i \in N\} \cup \{(d, (r, s'_i)) \mid i \in N\} \end{aligned}$$

PC

PC requires selections of an agent to reproduce and an agent to die, and a decision about whether the replacement will happen, so $V = \{d, r, p\}$. The selection of d is a uniformly random choice from the entire population, so it has no dependencies. r is a weighted random choice from $\Omega(d)$ proportional only to edge weight, yielding dependency $(d, r) \in D$.

Next, the decision is made about whether the replacement will occur, which depends on the fitnesses f_d and f_r, which in turn depend on $\{s_i \mid i \in \Omega(d) \cup \Omega(r)\}$. Thus, we have dependencies $\forall i \in N : (s_i, p) \wedge (d, (s_i, p)) \wedge (r, (s_i, p))$.

Finally, $\forall i \in N : s'_i$ is set to s_r if p or s_i otherwise, yielding dependencies $\forall i \in N : (s_i, s'_i) \wedge (p, s'_i) \wedge (d, (p, s'_i)) \wedge (d, (s_i, s'_i))$.

Thus, a single round of **PC** is represented by the following CTDG:

$$\begin{aligned}
T &= \{s_i \mid i \in N\} \cup \{s'_i \mid i \in N\} \cup \{d, r, p\} \\
D &= \{(d, r)\} \cup \{(r, p)\} \cup \{(s_i, p) \mid i \in N\} \\
&\quad \cup \{(d, (s_i, p)) \mid i \in N\} \cup \{(r, (s_i, p)) \mid i \in N\} \\
&\quad \cup \{(s_i, s'_i) \mid i \in N\} \cup \{(d, (s_i, s'_i)) \mid i, \in N\} \\
&\quad \cup \{(p, s'_i) \mid i \in N\} \cup \{(d, (p, s'_i)) \mid i \in N\}
\end{aligned}$$

BR

BR requires selections of an agent to die, and a computation of fitness maximization, so $V = \{d, f\}$. The selection of d is a uniformly random choice from the entire population, so it has no dependencies. f requires evaluating the fitness of d for every possible type, which depends on $\{s_i \mid i \in \Omega(d)\}$, yielding dependencies $\forall i \in N : (s_i, f) \in D \wedge (d, (s_i, f)) \in D \wedge (d, f) \in D$.

Finally, s'_d is set to the computed type f, thus $\forall i \in N : (f, s'_i) \in D \wedge (d, (f, s'_i)) \in D$.

Thus, a single round of **BR** is represented by the following CTDG:

$$\begin{aligned} T &= \{s_i \mid i \in N\} \cup \{s'_i \mid i \in N\} \cup \{d, f\} \\ D &= \{(d, f)\} \cup \{(s_i, f) \mid i \in N\} \cup \{(d, (s_i, f)) \mid i \in N\} \\ &\quad \cup \{(f, s'_i) \mid i \in N\} \cup \{(d, (f, s'_i)) \mid i \in N\} \end{aligned}$$

BDD

BDD requires selections of an agent to reproduce and an agent to die, so $V = \{d, r\}$. The selection of r is a uniformly random choice from the entire population, so it has no dependencies. d is a weighted random choice from $\Omega(r)$ proportional to fitness and edge weight, so it depends on $\forall i \in \Omega(r) : s_i$. This yields dependencies $(r, d) \in D$ and $\forall i \in N : (s_i, d) \in D \land (r, (s_i, d)) \in D$.

Finally, $\forall i \in N : s'_i$ is set to s_d if $r = i$ or s_i otherwise, yielding dependencies $\forall i \in N : (s_i, s'_i) \in D \land (d, s'_i) \in D \land (r, (s_i, s'_i)) \in D \land (r, (d, s'_i)) \in D$.

Thus, a single round of **BDD** is represented by the following CTDG:

$$\begin{aligned} T &= \{s_i \mid i \in N\} \cup \{s'_i \mid i \in N\} \cup \{d, r\} \\ D &= \{(r, d)\} \cup \{(s_i, d) \mid i \in N\} \cup \{(r, (s_i, d)) \mid i \in N\} \\ &\quad \cup \{(s_i, s'_i) \mid i \in N\} \cup \{(r, (s_i, s'_i)) \mid i, \in N\} \\ &\quad \cup \{(d, s'_i) \mid i \in N\} \cup \{(r, (d, s'_i)) \mid i \in N\} \end{aligned}$$

Appendix B

Resolvability

BD

Theorem B.0.1. ***BD*** *is neither input- nor output-resolvable.*

Proof. By inspection of $\langle O \cup V, D\rangle$, $\nexists s : \langle \sigma_0, R, U\rangle \Rightarrow s$. Further, $\exists(t_1, (s_i, t_2)) \in U \wedge \exists(t_1, (t_2, s_i')) \in U$. Thus, no computations can be done, but there are unresolved input and output dependencies, so **BD** is neither input- nor output- resolvable. □

DB

Theorem B.0.2. ***DB*** *is both input- and output-resolvable.*

Proof. By inspection of $\langle O \cup V, D\rangle$, we have $\langle \sigma_0, D, \emptyset\rangle \Rightarrow \langle \sigma_0[d], U, R\rangle$. In this state, $\nexists(t_1, (s_i, t_2)) \in U \wedge \nexists(t_1, (t_2, s_i')) \in U$. Thus, **DB** is both input- and output-resolvable. □

PC

Theorem B.0.3. ***PC*** *is both input- and output-resolvable.*

Proof. By inspection of $\langle O \cup V, D \rangle$, we have $\langle \sigma_0, D, \emptyset \rangle \Rightarrow \langle \sigma_0[d], U, R \rangle$. In this state, $\nexists (t_1, (s_i, t_2)) \in U \wedge \nexists (t_1, (t_2, s'_i)) \in U$. Thus, **DB** is both input- and output-resolvable. $\square$

BR

Theorem B.0.4. ***BR** is both input- and output-resolvable.*

Proof. By inspection of $\langle O \cup V, D \rangle$, we have $\langle \sigma_0, D, \emptyset \rangle \Rightarrow \langle \sigma_0[d], U, R \rangle$. In this state, $\nexists (t_1, (s_i, t_2)) \in U \wedge \nexists (t_1, (t_2, s'_i)) \in U$. Thus, **DB** is both input- and output-resolvable. $\square$

BDD

Theorem B.0.5. ***BDD** is input- but not output-resolvable.*

Proof. By inspection of $\langle O \cup V, D \rangle$, we have $\langle \sigma_0, D, \emptyset \rangle \Rightarrow \langle \sigma_0[r], U, R \rangle$. In this state, $\nexists (t_1, (s_i, t_2)) \in U$. However, $\exists (t_1, (t_2, s'_i)) \in U$. Thus, **BDD** is input- but not output-resolvable. $\square$

DBD

Theorem B.0.6. ***DBD** is neither input- nor output-resolvable.*

Proof. By inspection of $\langle O \cup V, D \rangle$, $\nexists s : \langle \sigma_0, R, U \rangle \Rightarrow s$. Further, $\exists (t_1, (s_i, t_2)) \in U \wedge \exists (t_1, (t_2, s'_i)) \in U$. Thus, no computations can be done, but there are unresolved input and output dependencies, so **DBD** is neither input- nor output- resolvable. $\square$

Bibliography

[1] Benjamin Allen and Martin Nowak. "Games on graphs". In: *EMS Surveys in Mathematical Sciences* 1.1 (2014), pp. 113–151. ISSN: 2308-2151. DOI: 10.4171/EMSS/3. URL: http://www.ems-ph.org/doi/10.4171/EMSS/3.

[2] Benjamin Allen et al. "How mutation affects evolutionary games on graphs." In: *Journal of theoretical biology* 299 (Apr. 2012), pp. 97–105. ISSN: 1095-8541. DOI: 10.1016/j.jtbi.2011.03.034. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3150603%5C&tool=pmcentrez%5C&rendertype=abstract.

[3] K. Burrage. "Parallel Implementation of Stochastic Simulation for Large-scale Cellular Processes". In: *Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)* (2005), pp. 621–626. DOI: 10.1109/HPCASIA.2005.67. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1592332.

[4] Krishnendu Chatterjee, Johannes G Reiter, and Martin a Nowak. "Evolutionary dynamics of biological auctions." In: *Theoretical population biology* 81.1 (Feb. 2012), pp. 69–80. ISSN: 1096-0325. DOI: 10.1016/j.tpb.2011.11.003. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3279759%5C&tool=pmcentrez%5C&rendertype=abstract.

[5] Christoph Hauert. "Effects of Space in 2fffdfffd2 Games". In: *International Journal of Bifurcation and Chaos* 12.7 (2002), pp. 1531–1548. ISSN: 0218-1274. DOI: 10.1142/S0218127402005273. URL: http://www.worldscientific.com/doi/abs/10.1142/S0218127402005273.

[6] Vipin Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co Inc., 1994.

[7] Erez Lieberman, Christoph Hauert, and Martin A Nowak. "Evolutionary dynamics on graphs". In: 433.JANUARY (2005). DOI: 10.1038/nature03211.1..

[8] Martin A Nowak. *Evolutionary Dynamics: Exploring the Equations of Life*. Harvard University Press, 2006.

[9] Hisashi Ohtsuki et al. "A simple rule for the evolution of cooperation on graphs and social networks." In: *Nature* 441.7092 (May 2006), pp. 502–5. ISSN: 1476-4687. DOI: 10.1038/nature04605. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2430087%5C&tool=pmcentrez%5C&rendertype=abstract.

[10] Jorge Pacheco, Arne Traulsen, and Martin Nowak. "Coevolution of Strategy and Structure in Complex Networks with Dynamical Linking". In: *Physical Review Letters* 97.25 (Dec. 2006), p. 258103. ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.97.258103. URL: http://link.aps.org/doi/10.1103/PhysRevLett.97.258103.

[11] David G Rand et al. "Static network structure can stabilize human cooperation." In: *Proceedings of the National Academy of Sciences of the United States of America* D (Nov. 2014). ISSN: 1091-6490. DOI: 10.1073/pnas.1400406111. URL: http://www.ncbi.nlm.nih.gov/pubmed/25404308.

[12] Carlos P. Roca, José a. Cuesta, and Angel Sánchez. "Effect of spatial structure on the evolution of cooperation". In: *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* 80 (2009), pp. 1–16. ISSN: 15393755. DOI: `10.1103/PhysRevE.80.046106`. arXiv: `arXiv:0806.1649v4`.

[13] F. C. Santos and J. M. Pacheco. "Scale-free networks provide a unifying framework for the emergence of cooperation". In: *Physical Review Letters* 95.August (2005), pp. 1–4. ISSN: 00319007. DOI: `10.1103/PhysRevLett.95.098104`.

[14] B. K. Sarkar, Shahid Jamal, and Bhagirath Kumar. "An efficient parallel algorithm for finding the largest and the second largest elements from a list of elements". In: *Proceedings - 9th International Conference on Information Technology, ICIT 2006* (2007), pp. 269–272. DOI: `10.1109/ICIT.2006.27`.

[15] Paulo Shakarian, Patrick Roos, and Anthony Johnson. "A review of evolutionary graph theory with applications to game theory." In: *Bio Systems* 107.2 (Feb. 2012), pp. 66–80. ISSN: 1872-8324. DOI: `10.1016/j.biosystems.2011.09.006`. URL: `http://www.ncbi.nlm.nih.gov/pubmed/22020107`.

[16] György Szabó and G Fath. "Evolutionary games on graphs". In: *Physics Reports* 446.4-6 (July 2007), pp. 97–216. DOI: `10.1016/j.physrep.2007.04.004`. arXiv: `0607344 [cond-mat]`. URL: `http://arxiv.org/abs/cond-mat/0607344%20http://www.sciencedirect.com/science/article/pii/S0370157307001810`.

[17] Matthijs van Veelen et al. "Direct reciprocity in structured populations." In: *Proceedings of the National Academy of Sciences of the United States of America* 109.25 (June 2012), pp. 9929–34. ISSN: 1091-6490. DOI: `10.1073/pnas.1206694109`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3382515%5C&tool=pmcentrez%5C&rendertype=abstract`.