5. Параллельные методы

Использование метаэвристик может потребовать больших ресурсов. Например, не принято использовать более 100 000 вычислений целевой функции за один запуск в генетическом программировании (скажем, для популяции из 2000 особей, запуск будет длиться 50 поколений). Оценка особей может длиться весьма долго: это может быть запуск модели, либо анализ структуры сложного химического соединения. В силу этого параллельные методы очень привлекательно выглядят.

На мой взгляд, это самый сильный аргумент в пользу использования параллельных методов. Однако ряд ученых считают, что некоторые параллельные методы (в частности, Островная модель, обсуждаемая в Разделе) уже сами по себе оказывают положительный эффект на процесс оптимизации. К примеру, Збигнев Сколички[1] (*Zbigniew Skolicki*) обнаружил функции приспособленности, в которых параллельные методы работали лучше, чем одна эволюционирующая популяция, даже если не учитывать ускорение, полученное от одновременного испольнения на нескольких компьютерах.

Многие методы стохастической оптимизации могут быть распараллелены, хотя для некоторых это сделать проще, чем для других. Методы с одним состоянием (локальный поиск, имитация отжига, поиск с запретом и др.) можно распараллелить, но, по моему мнению, очень неестественным образом. Вероятно, наиболее готовыми к параллельному исполнению являются методы, использующие популяцию решений, поскольку в них изначально одновременно рассматривается множество потенциальных решений, каждое из которых необходимо оценить. Пять основных методов распараллеливания[2]:

- Исполнять параллельно несколько запусков.
- Исполнять один запуск, в котором этап вычисления приспособленности (и возможно размножения и инициализации) обсчитывается в **несколько потоков** на одном компьютере.
- Исполнять раздельные запуски, между которыми время от времени осуществляется обмен приспособленными особями (распространение добра). Эти методы называют **Островной моделью** (***Island Models***).
- Исполнять один запуск, в котором на этапе вычисления приспособленности эта задача распределяется по удаленным компьютерам. Такой подход известен как оценивание приспособленности **Хозяин-Раб** (***Master-Slave***) или **Клиент-Сервер** (***Client-Server***).
- Исполнять один запуск с процедурой селекции, предполагающей, что особи записаны в параллельный массив на векторном компьютере (такие модели называются **пространственно вложенными** (***spatially embedded***) или **мелкозернистыми** (***fine-grained***)).

Эти пять методов можно комбинировать множеством способов. К примеру, нет никаких причин, запрещающих использование островной модели, в которой на каждом острове применяется вычисление приспособленности по модели хозяин-раб.

Пул потоков Ряд нижеперечисленных алгоритмов предполагают, что потоки уже порождены и зарегистрированы в едином *пуле потоков* (*thread pool*), из которого их можно вытаскивать и запускать. По их завершении мы снова вовращаем эти потоки в общий пул.

---

[0] Перевод раздела из книги Luke S. Essentials of Metaheuristics. A Set of Undergraduate Lecture Notes. Zeroth Edition. Online Version 1.2. July, 2011 (`http://cs.gmu.edu/~sean/book/metaheuristics/`). Перевел – Юрий Цой, 2011 г. Любые замечания, касающиеся перевода, просьба присылать по адресу yurytsoy@gmail.com

Данный текст доступен по адресу: `http://qai.narod.ru/GA/meta-heuristics_5.pdf`

[1] Zbigniew Skolicki, 2007, An Analysis of Island Models in Evolutionary Computation, Ph.D. thesis, George Mason University, Fairfax, Virginia.

[2] Несмотря на то, что немало исследователей внесли свой вклад в перечисленные методы, технический отчет Джона Грефенстетта (*John Grefenstette*) от 1981 г. предвосхитил удивительно многое. Его алгоритм A описывает многопоточный метод Хозяин-Раб; алгоритмы B и C описывают Асинхронную Эволюцию (*Asynchronous Evolution*) (вариант модели Хозяин-Раб, представленная ниже); а алгоритм D описывает островные модели. Информация взята из публикации John Grefenstette, 1981, Parallel adaptive algorithms for function optimization, Technical Report CS-81-19, Computer Science Department, Vanderbilt Univesity.

Алгоритм 65 Функции пула потоков

1: **global** $l \leftarrow$ блокировщик пула.
2: **global** $T \leftarrow \{\}$ пустой пул для кортежей $\overrightarrow{t} = \langle t_{lock}, t_{data} \rangle$, где t_{lock} – блокировщик, а t_{data} – любой объект.

3: **procedure** InsertMyselfAndWait( )
4: Запрос блокировки l
5: $\overrightarrow{t} \leftarrow$ новый кортеж $\langle t_{lock}, t_{data} \rangle$ {t_{lock} – новый блокировщик, t_{data} на данный момент может быть чем угодно}
6: $T \leftarrow T \cup \{t\}$
7: Запрос блокировки t_{lock}
8: Оповещение потоков, ожидающих l
9: Ожидание t_{lock} {Здесь освобождаются оба блокировщика, ожидается оповещение для t_{lock}, а затем перезахватываются блокировки}
10: $o \leftarrow$ копия t_{data} {К этому моменту t_{data} присваивается в TellThreadToStart(...)}
11: Снятие блокировки t_{lock}
12: Снятие блокировки l
13: **return** o

14: **procedure** ThreadIsInserted( )
15: Запрос блокировки l
16: **if** T пуст **then**
17: Снятие блокировки l
18: **return** false
19: **else**
20: Снятие блокировки l
21: **return** true
22: **end if**

23: **procedure** TellThreadToStart(Информация o)
24: Запрос блокировки l
25: **while** T пуст **do**
26: Ожидание l {Здесь снимается блокировка, ожидается оповещения для l и происходит перезахват блокировщика}
27: **end while**
28: $t \leftarrow$ произвольный кортеж из T
29: $T \leftarrow T - \{t\}$
30: Запрос блокировки t_{lock}
31: $t_{data} \leftarrow$ копия o
32: Оповещение потоков, ожидающих t_{lock}
33: Снятие блокировки t_{lock}
34: Снятие блокировки l
35: **return** t

36: **procedure** WaitForAllThreads(количество потоков n)
37: Запрос блокировки l
38: **while** $\|T\| < n$ **do**
39: Ожидание l
40: **end while**
41: Снятие блокировки l

Эти алгоритмы выглядят сложно и их непросто отладить. К пулу потоков, в свою очередь, имеются следующие требования:

- Порождение потоков.
- *Запрос* и *снятие* блокировки для каждого потока. Если поток пытается запросить блокировку, которая уже занята, то он приостанавливается до тех пор, пока запрашиваемый блокировщик не будет освобожден.
- Возможность *ожидания* блокировки, означающая, что возможность освобождения блокировщика для других потоков и приостановку до тех пор, пока какой-нибудь поток не разошлет *оповещение* о захвате блокировщика.
- Возможность оповещения потоков, ожидающих некоторый блокировщик. В результате потоки по очереди получают блокировщик, возобновляют работу и делают, что хотят.

В принципе, любая многопоточная библитека предоставляет эти возможности. С их помощью можно породить сколько угодно потоков, и направить их путем вызова функции InsertMyselfAndWait, чтобы они получили особь, с которой должны работать. Это все стандартные процедуры, хотя и немного запутанные.

5.1 Множественные потоки

Для оценки приспособленности можно распределить особей по потокам, когда последние становятся доступными. Когда поток завершает работу с одной особью, он готов приступить к работе со следующей.

Алгоритм 66 Мелкозернистое параллельное оценивание приспособленности

```
1: P ← текущая популяция

2: for каждая особь P_i ∈ P do
3:    TellThreadToStart({P_i}) {Если всего потоков ≥ ‖P‖, то хотя бы один всегда будет доступен.}
4: end for
5: WaitForAllThreads()
6: return P
```

Данный алгоритм требует наличия пула потоков. В более простом подходе блокировка не требуется, популяция просто разбивается на группы, и каждая группа обрабатывается своим отдельно порожденным потоком. В конце результаты работы различных потоков сливаются вместе.

Алгоритм 67 Простое параллельное оценивание приспособленности

```
1: P ← популяция {P_1, ..., P_l}
2: T ← множество потоков {T_1, ..., T_n}

3: for i от 1 до n do
4:    a ← ⌊l/n⌋ × (i − 1) + 1 {поиск нижней (a) и верхней (b) границы для i группы особей}
5:    if i = n then
6:       b ← l
7:    else
8:       b ← ⌊l/n⌋ × i
9:    end if
10:   Породить поток T_i и вызвать в нем функцию оценки особей от P_a до P_b.
11: end for
12: for i от 1 до n do
13:   Ожидание завершения потока T_i
14: end for
15: return P
```

В данном случае будет необходима только возможность создания потоков и ожидания их завершения (и то, и другое – стандартные функции библиотеки для работы с потоками). Можно «ждать»

завершения потоков просто путем «слияния» (*join*) с ними (стандартная функция в параллельных библиотеках). Недостатков этого варианта является то, что в одну группу могут попасть особи, оценка которых производится дольше (если это возможно), и для завершения работы процедуры другие потоки будут простаивать, ожидая завершения этого самого медленного.

Подобный прием можно использовать и для инициализации популяции, хотя обычно инициализация не занимает много времени. Тем не менее, можно использовать алгоритм 66, только вместо оценки существующей особи P_i в каждом потоке будет создаваться новая особь и помещаться в популяцию в позицию i. Аналогично можно использовать для инициализации и алгоритм 67, в которой вместо функции ОценкаПриспособленности($P_a \dots P_b$), в каждом потоке будут проинициализированы $b - a + 1$ особей и помещены в позиции $a \dots b$.

Точно также можно распараллелить этап размножения, однако в данном случае могут возникнуть осложнения, вызванные выбором процедуры селекции, из-за чего некоторые вычисления необходимо сделать заранее. Однако турнирная селекция отлично подходит, и для нее не требуется никаких предварительных расчетов. Обратите внимание, что мы уже не делим популяцию P на группы, но группы появляются в популяции Q следующего поколения:

Алгоритм 68 Простое параллельное размножение как в генетическом алгоритме

1: $P \leftarrow$ текущая популяция
2: $T \leftarrow$ множество потоков $\{T_1, \dots, T_n\}$
3: $l \leftarrow$ требуемый размер новой популяции

4: $\overrightarrow{q} \leftarrow$ пустой массив $\langle q_1, \dots, q_l \rangle$ {Для хранения новых особей}
5: **for** i от 1 до n **do**
6: $a \leftarrow \lfloor l/n \rfloor \times (i-1) + 1$ {поиск нижней (a) и верхней (b) границы для i группы особей}
7: **if** $i = n$ **then**
8: $b \leftarrow l$
9: **else**
10: $b \leftarrow \lfloor l/n \rfloor \times i$
11: **end if**
12: Породить поток T_i и вызвать в нем функцию порождения особей от q_a до q_b.
13: **end for**
14: **for** i от 1 до n **do**
15: Ожидание завершения потока T_i
16: **end for**
17: **return** массив $\overrightarrow{q}$ преобразованный в популяцию

Алгоритм 69 Мелкозернистое параллельное размножение как в генетическом алгоритме

1: $P \leftarrow$ текущая популяция
2: $l \leftarrow$ требуемый размер новой популяции {Предполагается, что число – четное.}

3: $\overrightarrow{q} \leftarrow$ пустой массив $\langle q_1, \dots, q_l \rangle$ {Для хранения новых особей}
4: {Используется шаг 2 в предположении, что кроссинговер создает двух потомков.}
5: **for** i от 1 до l с шагом 2 **do**
6: TellThreadToStart($\{\overrightarrow{q}, i, i+1\}$) {Поток использует популяцию для создания потомков q_i и q_{i+1}.}
7: **end for**
8: WaitForAllThreads(общее число потоков)
9: **return** массив $\overrightarrow{q}$ преобразованный в популяцию

Причина, по которой эти алгоритмы нормально работают заключается в том, что они являются примерами процедур с **предварительным копированием** (***copy-forward***): из P выбираются особи, *копируются*, а затем модифицируются полученные *копии*. Поэтому нет необходимости беспокоиться о блокировке особей из P. Другие процедуры могут оказаться сложнее.

5.2 Островные модели

Островная модель представляет группу работающих одновременно процессов эволюционной оптимизации, которые время от времени пересылают друг другу особей для распространения инфор-

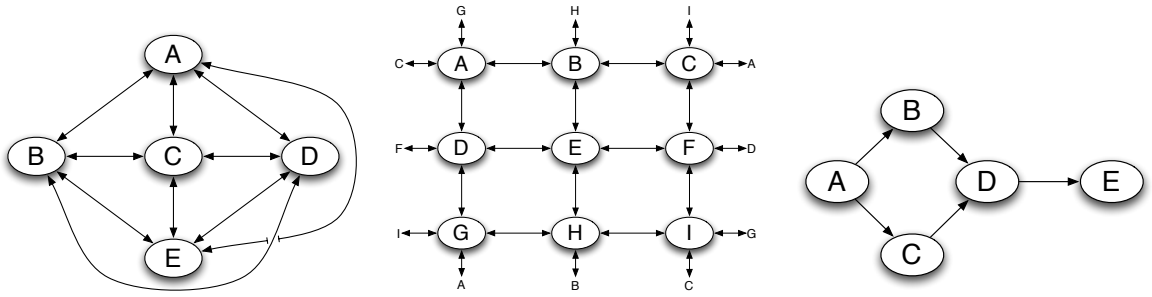

Рис. .1: Полносвязная, двумерная тороидальная сетка и инъекционная топологии для островной модели

мации о недавно обнаруженных областях с высокой приспособленностью. В некоторых островных моделях производится пересылка наиболее приспособленных особей, либо для выбора особи используется оператор селекции. В других для пересылки используются случайные особи (что соответствует меньшей эксплуатации).

Первоначально островные модели разрабатывались для реализации преимущества с точки зрения вычислительной эффективности. Было принято загружать каждый из n компьютеров своим потоком. Но у них есть и другая особенность: поскольку популяция разбивается на отдельные **подпопуляции** (***subpopulations***) (иногда называемые **демами** (***demes***)), то приспособленные особи медленнее распространяются в популяции, из-за чего в системе повышается разнообразие и сильнее исследовательская составляющая. Еще одним аргументом является и то, что если цеелвую функцию можно декомпозировать на несколько независимых компонент, то теоретически в островных моделях каждая из этих компонент может быть оптимизирована отдельной популяцией.

Для создания островной модели необходимо выбрать **топологию связи островов** (***island topology***). Нужно решить: с каких островов и на какие будут пересылаться особи? Общепринятыми являются три топологии, показанные на рис. .1. В **полносвязной** (***fully-connected***) топологии все острова попарно связаны. В топологии в виде **тороидальной сети** (***toroidal grid***), острова помещаются в узлы, хм, n-мерной тороидальной сети. При полносвязаной топологии для перемещения особи с одного острова на любой другой необходим один прыжок, но в сетевой топологии перемещение до удаленной популяции может занять много времени и включать различные промежуточные популяции. Поэтому в последнем случае вероятно большее разнообразие поиска.

Еще одна топология: **инъекционная модель** (***injection model***), – представлена структурой с прямым распространением. Иногда она используется для облегчения задачи. К примеру, необходимо создать робота, играющего в футбол. В ЭА не получается получить все и сразу, поэтому создается конвейер, в котором осуществляется поэтапное решение задачи. На первом острове целевая функция выглядит так: насколько хорошо робот пинает мяч? Затем особи мигрируют на следующий остров, в которых функция приспособленности уже изменилась: насколько хорошо робот отдает пас? Затем можно рассмотреть миграцию на остров, на котором целью является сохранение мяча вдали от противника, отдавая пасы членам команды. И так далее.

После того, как определена топология, необходимо модифицировать ЭА. Вот мой вариант:

В данном примере абстрактный ЭА дополнен метафорой почтового ящика: на каждом острове есть почтовый ящик, на который с других островов можно отправлять особей. На островах же можно, по мере необходимости, извлекать и использовать особей из ящиков. Таким образом с использованием данной метафоры к абстрактному алгоритму добавлены три новые функции: **отправка** особей на соседние острова, **получение** особей с соседних островов, оказавшихся в почтовом ящике, **объединение** полученных особей с текущей популяций (обратите внимание, что функция Join теперь имеет три аргумента). При использовании инъекционной модели, для особей, оказавшихся в почтовом ящике, возможно необходимо пересчитать приспособленность в соответствии с используемой на данном острове целевой функцией.

Можно сделать так, что ваша реализация будет использовать **синхронный** вариант алгоритма, в котором все острова ждут, пока все присланные особи не будут обработаны, а только потом делают новую рассылку. Но в большинстве случаев **асинхронный** вариант позволяет лучше использовать ресурсы сети. В нем особи просто отсылаются когда нужно, и ждут в ящике острова-получателя, пока он не будет готов их обработать. В результате некоторым островам можно работать медленнее, чем другим. Естественно, в этой ситуации необходимо решать, что делать, если почтовый ящик

Алгоритм 70 Абстрактный популяционный эволюционный алгоритм с системой сообщений по типу островной модели

```
1: P ← создание начальной популяции
2: Best ← □

3: repeat
4:    ВычислениеПриспособленности (P)
5:    Разослать копии некоторых особей из P по адресам на соседних островах
6:    for каждая особь P_i ∈ P do
7:       if Best = □ или Приспособленность(P_i) > Приспособленность(Best) then
8:          Best ← P_i
9:       end if
10:   end for
11:   Извлечь и вернуть всё содержимое почтового ящика
12:   P ← Join(P, M, Breed(P)) {Есть вероятность, что сгенерированные в результате размножения
      особи останутся невостребованными}
13: until Best – идеальное решение, либо истекло время поиска
14: return Best
```

переполнен.

Еще одним параметров, влияющим на пропускную способность сети является количество и тип связей в выбранной топологии. Какие компьютеры соединены, Как часто они делают рассылку особей друг другу и в какие моменты времени? Сколько особей пересылается за раз? Большое количество связей или слабосвязаные топологии могут вызывать нагрузку на отдельные части сети. На моей кафедре имеется кластер с двумя сетями, и в каждой есть свой маршрутизатор. Два маршрутизатора связаны между собой быстрым каналом, но его скорости недостаточно. Поэтому мне, наверное, бы хотелось иметь конфигурацию сети, в которой узлы, относящиеся к одному маршрутизатору, могли бы общаться значительно чаще. В дополнение я могу сконфигурировать острова так, чтобы они рассылал особей каждые m поколений, и тогда остров, занимающийся отправкой, впадает в ступор. Планируя делать подобные конфигурации, подумайте о том, чтобы максимизировать пропускную способность.

В разделе было детально показано, как обрабатывать блокировки и т.д., необходимые для параллельных моделей. Для островных моделей подобный разбор не приводится, однако здесь нет ничего супер сложного. Соседние острова соединяются через сокеты, затем используется либо функция UNIX select(), либо создается отдельный поток для каждого сокета. Остановимся на последнем варианте. Поток в цикле читает сокет, блокирует почтовый ящик, добавляет данные в ящик и снимает блокировку. В основном ЭА для получения доступа к текущему содержимому ящика и его очистки необходимо сначала его заблокировать, а после чтения – разблокировать. Отправка особей на соседние острова осуществляется просто через запись в соответствующие удаленные сокеты (без использования потоков).

Вычисление приспособленности по схеме хозяин-раб

Это наиболее общая форма параллельных метаэвристик, и к тому же самая очевидная. Имеющиеся компьютеры делятся на **хозяина** (***master***) и n **рабов** (***slaves***)[3]. При оценке особи, она отправляется к рабу, чтобы он работал. Такой подход становится более полезным, когда увеличивается время оценки приспособленности особей. А для огромного количества задач оптимизации, представляющих интерес в настоящее время, время оценки приспособленности может быть настолько большим, что оно безоговорочно является доминирующим фактором, определяющим длительность процесса поиска решения.

Итак, как задать схему хозяин-раб? Если по-простому, то также, как осуществляется многопоточное оценивание приспособленности (алгоритмы 65 и 66, я бы не стал обращать внимания на алгоритм 67). Единственная разница заключается в том, что каждый поток, зарегистрированный в общем пуле, привязан к определенному работу через сокет. Вместо оценки особи (особей) непосредственно в потоке, поток пересылает их в сокет для удаленной обработки. Для этого необходимо

[3] Или, если угодно, *клиент* (***client***) и *серверы* (***servers***). Или лучше *сервер* и *клиенты*?

модифицировать алгоритм 66, чтобы разрешить потоку получать сразу несколько особей, например, так:

Алгоритм 71 Мелкозернистое оценивание приспособленности по схеме хозяин-раб

1: $P \leftarrow$ текущая популяция $\{P_1, \ldots, P_l\}$
2: $n \leftarrow$ количество особей, пересылаемых рабу за 1 раз

3: **for** i от 1 до l с шагом n **do**
4: TellThreadToStart($\{P_i, \ldots, P_{\min(i+n-1,l)}\}$) {Отправка особей к удаленному рабу.}
5: **end for**
6: WaitForAllThreads()
7: **return** P

Такой подход весьма неплох, т.к. он относительно щадяще относится к медленным рабам и колебаниям времени оценивания приспособенности, а также позволяет одновременно работать нескольким рабам. Обработка рабов, которые прекращают работу в процессе оценки приспособленности потребуются более сложные методы, которые мы не будем рассматривать (хотя они важны!).

Когда схема хозяин-раб полезна? Все зависит от размеров сети и скорости передачи данных. Подход по схеме хозяин-раб приносит пользу, когда имеется необходимое количество компьютеров и достаточная скорость передачи данных, чтобы время, затрачиваемое на пересылку особей и возврат результата, было меньше, чем время оценки приспособленности на одном процессоре. Есть пара приемов, благодаря которым можно увеличить пропускную способность. По-первых, можно сжимать особей, содержащих большой объем лишней информации. Во-вторых, во многих случаях нет необходимости возвращать обратно особь от раба: необходимо только приспосоленность (хотя бывает по-разному, как мы вскоре убедимся). В-третьих, во многих сетях данные пересылаются пакетами с постоянным размером служебной информации, поэтому более эффективным является заполнение пакета по максимуму, если это возможно. Отправка связки особей к рабу, может хорошо заполнить пакет, в то время как при отправке одной особи будет потеря эффективности.

Вот как, на мой взгляд, относятся между собой схема хоязин-раб и островная модель. Если пропускная способность сети низкая и время оценки приспособленности очень мало, то лучше выбирать островную модель. Также лучше использовать островную модель, если необходимо обработать *очень* большую популяцию. В остальных случаях я предпочитаю схему хозяин-раб.

Оба подхода можно смешивать. К примеру, можно иметь набор островов, для каждого из которых имеется свое множество рабов, выполняющих оценку приспособленности. Либо можно использовать то, что я называю **Оппортунистской Эволюцией**[4] (***Opportunistic Evolution***), которая может хорошо подходить для грид-вычислений. В этом подходе имеется множество рабов, как обычно, и каждому из них отправляется n особей. Каждому рабу также предоставляется определенное довольно большое время, достаточное для того, чтобы оценить приспособленность особей. Если раб заканчивает вычисления, и у него остается время из отведенного интервала, то раб производит небольшую оптимизацию (возможно, локальный или эволюционный поиск) для имеющейся минипуляции из n особей. Когда время заканчивается, раб возвращает хозяину вместо исходной обновленную минипопуляцию. (Заметьте, что в данном случае *необходимо* отправлять особей обратно, а не только приспособленность.)

Можно также весьма элегантно комбинировать оценивание приспособленности по схеме Хозяин-Раб с устойчивым генетическим алгоритмом, что получило название **Асинхронной эволюции** (***Asynchronous Evolution***). Как только раб готов получить особей для оценки, производится селекция и генерация потомков, которые пересылаются рабу. В асинхронной эволюции не требуется ожидание завершения работы всех рабов – она асинхронная, – вместо этого по мере окончания работы рабом его оосби присоединяются к популяции. Различным рабам требуется различное время для работы. Этот подход не страдает даже от *чрезвычайно* сильного разброса времени вычисления приспособленности, что вполне естественно, т.к. особи, на оценку которых уходит много времени, реже размножаются.

[4] Это понятие был впервые предложено в техническом отчете: Ricardo Bianchini and Christopher Brown, 1993, Parallel genetic algorithms on distributed-memory architectures, Revised Version 436, Computer Science Department, University of Rochester, Rochester, NY 14627.

Термин *оппортунистская эволюция* был введен Стивеном Арментраутом (*Steven Armentrout*) в нашей статье: Keith Sullivan, Sean Luke, Curt Larock, Sean Cier, and Steven Armentrout, 2008, Opportunistic evolution: efaшcient evolutionary computation on large-scale computational grids, in *GECCO'08: Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation*, pages 2227–2232, ACM, New York, NY, USA.

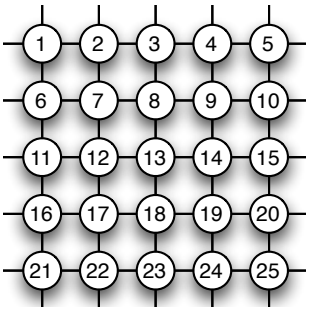

Рис. .2: Пространственно-вложенная популяция особей на двухмерной сетке. Показаны особи с 1-ой по 25-ую.

В асинхронной эволюции используется потокобезопасный набор (мультимножество). Когда поток получается особей от удаленного раба, этот поток вставляет особей в популяцию используя функцию AddToCollection (...). Алгоритм асинхронной эволюции запрашивает новых особей путем вызова функции RetrieveAllFromColection (...). Реализуется элементарно:

Алгоритм 72 Потокобезопасные функции для наборов

1: **global** $S \leftarrow \{\}$
2: **global** $l \leftarrow$ блокировщик для набора S

3: **procedure** AddToCollection(...)
4: Запрос блокировки l
5: $S \leftarrow S \cup T$
6: Снятие блокировки l

7: **procedure** RetrieveAllFromCollection()
8: Запрос блокировки l
9: $T \leftarrow S$
10: $S \leftarrow \{\}$
11: Снятие блокировки l
12: **return** T

При наличии потокобезопасного набора алгоритм асинхронной эволюции работает следующим образом:

Пространственно-вложенные модели

В пространственно-вложенной модели к популяции добавляется параметр *физического расположения* (*physical location*) особей. К примеру, популяция может быть расположена на трехмерной сетке, или на одномерном кольце, где каждая особь занимает определенную точку в пространстве. На рис. .2 изображены особи на двухмерной сетке.

Такие модели используются в основном для поддержания разнообразия в популяции и поддержке исследования пространства поиска. Особи могут размножаться только с «соседними» особями, поэтому высокоприспособленные особи не смогут так быстро распространиться в популяции, как если бы ограничения на скрещивание отсутствовали. Образно выражаясь области в расположении популяции могут развивать свою культуру, так сказать, по аналогии с островными моделями.

Пространственно-вложенные модели можно распараллелить в многопоточном смысле. Однако при наличии **векторного процессора** (***vector processor***), т.е. вычислительного устройства, производящего множество идентичных одновременных операций в единицу времени, существуют методы выполнения скрещивания и селекции на одном таком процессоре. В настоящее время наиболее распространенным векторным процессором, который может оказаться в вашем распоряжении, является графический процессор (ГП, *GPU*). Предполагая, что используется многопоточный пример,

Алгоритм 73 Асинхронная эволюция

```
1: P ← {}
2: n ← количество особей, пересылаемых рабу за 1 раз
3: popsize ← требуемый размер популяции

4: Best ← □
5: repeat
6:   if ThreadIsInserted( ) = true then
7:     {Если есть простаивающие процессоры.}
8:     if ‖P‖ < popsize then
9:       Q ← n случайных особей {Инициализация не окончена.}
10:    else
11:      Q ← {}
12:      for i от 1 до n с шагом 2 do
13:        {Естественно, можно использовать и другой способ размножения.}
14:        Родитель P_a ← ВыборСЗамещением (P);
15:        Родитель P_b ← ВыборСЗамещением (P);
16:        Потомки C_a, C_b ← Кроссинговер (Копия(P_a), Копия(P_b));
17:        Q ← Q∪{Мутация (C_a), Мутация (C_b)}
18:      end for
19:    end if
20:    TellThreadToStart({Q_i, ..., Q_n})
21:  end if
22:  M ← RetrieveAllFromCollection() {Получить всех особей, для которых вычисление приспособ-
     ленности завершено}
23:  for каждая особь M_i ∈ M do
24:    if Best = □ или Приспособленность(M_i) > Приспособленность(Best) then
25:      Best ← M_i
26:    end if
27:    if ‖P‖ = popsize then
28:      Особь P_d ← ВыбратьДляУничтожения(P) {Усточивость (steady state).}
29:      P ← P − {P_d}
30:    end if
31:    P ← P ∪ {M_i}
32:  end for
33:  if ThreadIsInserted() = false и M пусто then
34:    Небольшая пауза {Ничего не происходит: Дайте отдых ЦП.}
35:  end if
36: until Best – идеальное решение, либо закончилось время
37: return Best
```

можно легко реализовать многопоточное вычисление приспособленности и многопоточное размножение. Нужно просто модифицировать процедуру размножения.

Алгоритм 74 Пространственное размножение

$P \leftarrow$ текущая популяция, расположенная в пространстве

$Q \leftarrow$ новая популяция, расположенная также, как и P
for каждая особь $P_i \in P$ **do**
  {Можно организовать вычисления в несколько параллельных потоков, как и ранее}
  $N \leftarrow$ Соседи(P_i, P)
  Родитель $N_a \leftarrow$ ВыборСЗамещением(N)
  Родитель $N_b \leftarrow$ ВыборСЗамещением(N)
  Потомки $C_a, C_b \leftarrow$ Кроссинговер (Копия(N_a), Копия(N_b));
  $Q_i \leftarrow$ {Мутация (C_a) {C_b отбрасывается. C_a помещается в пространственный слот i в Q}
end for
return Q

Можно, если так больше нравится, делать только мутацию без кроссинговера. Важно то, что особи заменяются особым образом для каждого слота с использованием его соседей.

Для замены особи P_i выполняется селекция, но не по всей популяции, а по подмножеству N **соседей** P_i. Функция принадлежности к окрестности выбирается самостоятельно. Можно определять соседей P_i как особей, попадающих в параллелограмм с центром в P_i и протяженностью m в каждом направлении. Или можно формировать N совершая случайные блуждания, начиная от точки P_i. В конце каждого блуждания особь, находящаяся в текущем узле, помещается в N. Выбираемые подобным образом особи определяются с использованием гауссо-подобного распределения с центром в P_i. Чем длиннее блуждание, тем большое размер окрестности. Например:

Алгоритм 75 Селекция случайным блужданием

$P \leftarrow$ текущая популяция
$r \leftarrow$ выбранная длина случайного блуждания
$P_i \leftarrow$ начальная особь

$\overrightarrow{l} \leftarrow$ расположение $\langle l_1, \ldots, l_n \rangle$ особи P_i в пространстве
for r раз **do**
  repeat
    $d \leftarrow$ случайное целое, равномерно распределенное в интервале от 1 до n
    $j \leftarrow$ либо 1, либо -1, выбирается случайно
    if $l_d + j$ принимает допустимое значения для измерения d в пространстве **then**
      $l_d \leftarrow l_d + j$ {При необходимости, в тороидальном пространстве, необходимо «завернуть» координаты}
    else
      break {Покинуть цикл.}
    end if
  until *true*
end for
return Особь в узле $\overrightarrow{l}$ пространства.