# ОБЗОР МЕТОДОВ АВТОМАТИЗАЦИИ МОНИТОРИНГА, АНАЛИЗА И ВИЗУАЛИЗАЦИИ ПОВЕДЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ, ВЗАИМОДЕЙСТВУЮЩИХ С ПОМОЩЬЮ ПЕРЕДАЧИ СООБЩЕНИЙ

*М.А. Посыпкин, А.А. Соколов*

**Аннотация.** Обзор посвящен основным теоретическим концепциям, алгоритмам и методам автоматизированного анализа поведения параллельных программ, направленным на выявление ошибочных ситуаций. Также рассматриваются различные инструментальные средства, поддерживающие такой анализ. Производится сравнение существующих подходов и программных продуктов и делается общий вывод о состоянии дел в вопросах автоматизированной отладки параллельных программ.

## *1. Введение*

Тестирование и отладка являются неотъемлемыми частями процесса разработки программного обеспечения. Основной задачей, которая решается на этих фазах, является обнаружение и выяснение причин ошибочного поведения. Для решения этой задачи, как правило, применяются следующие методы:

- **Статический анализ.** Статический анализ применяется для того, чтобы на этапе трансляции указать один или последовательность операторов в тексте программы, выполнение которых может стать причиной некорректного поведения.
- **Динамический и “посмертный” анализ.** К этой категории относятся методы анализа информации (трассы), получаемой во время выполнения программы, применяемые в процессе или после завершения ее работы. Автоматизированный или непосредственный анализ собранной информации позволяет выявить аномалии в поведении программы и, в ряде случаев, также найти их причину.
- **Интерактивная отладка.** Этот метод заключается в том, что пользователь запускает программу под управлением специального приложения, называемого отладчиком, которое предоставляет возможности для отображения и управления состоянием выполняемой программы. Отладчик существенно облегчает задачу выяснения причины ошибки.
- **Тестирование.** Тестирование состоит в том, чтобы построить максимально полную в соответствии с некоторым критерием



(покрытия) систему тестов, которая позволит либо удостовериться в работоспособности программы, либо обнаружить ошибку.

Реализация перечисленных методов для случая параллельных программ сталкивается с рядом сложностей, обусловленных параллельной природой приложения. Можно выделить две основных причины этих сложностей. Первая причина состоит в усложнении структуры состояния, которое в случае параллельной программы определяется совокупностью состояний составляющих ее процессов и каналов взаимодействия между ними. Вторая причина заключается в усложнении семантики выполнения программы, состоящем в появлении взаимодействия процессов. Рассмотрим подробнее сложности, возникающие при реализации основных методов отладки и тестирования параллельных программ.

Применение традиционных методов статического анализа, предназначенных для выявления ошибок в последовательных программах, в общем случае не позволяет выявить ошибки взаимодействия и другие аномалии, характерные для параллельных программ. Сама задача статического анализа становится существенно более сложной из-за усложнения семантики параллельной программы.

Динамический и посмертный анализ усложняются за счет возрастающего объема трассы, т.к. трассировке подвержены несколько процессов и каналов взаимодействия. Фаза анализа трассы также становится более сложной, поскольку требуется анализировать и визуализировать одновременно несколько трасс, состоящих из событий, связанных между собой за счет взаимодействия между процессами.

Эффективная реализация интерактивной отладки связана с рядом трудностей. Во-первых, это – возросшая по сравнению с последовательной программой сложность отображения состояния параллельной программы. Во-вторых, управление поведением параллельной программы также усложняется, так как понятия «точка останова» и «шаг» для параллельной программы определяются сложнее, чем для последовательной. В-третьих, недетерминизм, характерный для параллельных программ, требует дополнительных усилий для фиксации и воспроизведения хода выполнения приложения.

Главная сложность, связанная с тестированием параллельных программ, также обусловлена присущим им недетерминизмом. Тестирование последовательного приложения состоит в создании множества наборов входных данных, проверяющих различные аспекты функциональности тестируемого приложения. В случае недетерминированной программы необходимо осуществлять перебор вариантов недетерминированного поведения. Кроме этого, критерии покрытия, обычно применяемые для последовательных программ, требуют расширения для случая параллельных приложений за счет учета взаимодействий между процессами.

Таким образом, задача эффективной реализации методов тестирования и отладки параллельных приложений является достаточно сложной и не может быть решена механическим перенесением подходов, разработанных для случая

последовательных программ. В данном обзоре рассматриваются методы и программные инструменты для решения следующих проблем из числа перечисленных выше:

- Трассировка параллельной программы, минимизация трассы, повторное воспроизведение хода выполнения программы по собранной трассе.
- Адекватная, масштабируемая визуализация хода выполнения параллельной программы по ее трассе.
- Автоматизация динамического и посмертного анализа параллельных программ на основе трассировочной информации.
- Перебор вариантов выполнения недетерминированный параллельной программы.

Тематика данного обзора ограничена также только рассмотрением параллельных программ, состоящих из нескольких параллельных процессов, взаимодействующих с помощью передачи сообщений. К этому разряду можно отнести программы, выполняющиеся в средах параллельного программирования MPI [1], PVM [2] и других библиотек, поддерживающих взаимодействие процессов с помощью передачи сообщений. Далее в статье термин параллельные программы применяется только для обозначения таких программ.

## *2. Основные источники ошибочного поведения в параллельных программах*

Как уже отмечалось, параллельные программы обладают рядом особенностей, которые принципиально отличают их от последовательных программ и существенно затрудняют процесс отладки и тестирования. Далее в этом разделе рассматриваются некоторые из этих особенностей, а также содержание и пути решения связанных с ними проблем.

### 2.1. Тупиковые ситуации в параллельных программах

*Тупиковая ситуация (deadlock)* возникает, когда один или несколько процессов ожидают выполнения условия, которое никогда не становится истинным [3].

Одна из наиболее распространенных причин тупиковой ситуации – *взаимная блокировка* [4] параллельных процессов. Она возникает тогда, когда один из процессов не может послать данные другому процессу, потому что тот ожидает сообщение от него.

Тупиковые ситуации могут также иметь место при выполнении операций коллективного взаимодействия – пересылки данных, в которую вовлечены сразу несколько процессов. При этом тупиковая ситуация возникает если не все процессы, которые должны участвовать в коллективной операции, выполняют соответствующую команду.

Тупиковые ситуации обычно проявляются как зависание программы. Их обнаружение представляет значительные трудности и требует специальной



поддержки, которая бы позволяла выявлять такие ошибки в процессе выполнения или отладки и сообщать пользователю об их возникновении.

### 2.2. Недетерминированное поведение параллельных программ

Любой разработчик программного обеспечения сталкивался с ситуацией, когда ошибка в программе происходит не при каждом запуске, или с ситуацией, когда от запуска к запуску программа выдает различные результаты. Такие программы обычно называют *недетерминированными*.

Дать строгое определение недетерминированной программы достаточно сложно. Неформально под детерминированными можно понимать программы, эффект от выполнения которых в системе определяется только входными данными программы, а под недетерминированными – все прочие программы.

Другими словами, наблюдаемое поведение детерминированных программы не должно зависеть от каких-либо внешних по отношению к программе факторов, отличных от входных данных программы.

В строгом смысле, детерминированных программ не существует, т.к. всегда можно найти внешнее воздействие, которое повлияет на результат выполнения программы. Примером такого воздействия может быть отключение или физическое разрушение компьютера в момент выполнения программы. Поэтому, представляется более правильным понимать детерминизм программы как независимость ее поведения от некоторого числа факторов, которые можно охарактеризовать как множество допустимых внешних воздействий. Множество допустимых внешних воздействий зависит от требований, предъявляемых программе. Например, для устойчивых к сбоям систем, в число допустимых воздействий может быть включено внезапное выключение компьютера, выход из строя одного из узлов вычислительного комплекса или части сетевого оборудования. Для прикладных параллельных программ к допустимым воздействиям относятся такие факторы, как загруженность процессорных устройств и каналов связи вычислительной системы, действия операционной системы, связанные с планированием выполнения процессов параллельной программы, задержки, возникающие при передаче сообщений. Естественно, пользователь стремится к тому, чтобы результат работы программы не зависел от перечисленных факторов.

Параллельное приложение состоит из нескольких параллельных процессов. Недетерминизм в таких приложениях наблюдается в связи с тем, что различные относительные скорости работы и взаимодействия процессов, а также объем доступной памяти для буферизации сообщений приводят к тому, что взаимодействие протекает по различным сценариям для разных запусков программы с одними и теми же входными данными. Это обусловлено особенностями функций, предназначенных для обмена сообщениями между процессами. Рассмотрим эти особенности более подробно на примере MPI – наиболее распространенной библиотеки для параллельного программирования.

Для отправки сообщения используется функция MPI_Send, а для приема – соответственно функция MPI_Recv. Определения этих функций приведены на Рис. 1.

**int MPI_Send( buf, count, datatype, dest, tag, comm )**
**void *buf** – буфер посылаемых данных
**int count** – число посылаемых элементов данных
**int dest** – номер процесса-получателя
**int tag** – тэг посылаемого сообщения
**MPI_Datatype datatype** – тип посылаемых элементов данных
**MPI_Comm comm** - коммуникатор

**int MPI_Recv( buf, count, datatype, source, tag, comm, status )**
**void *buf** – буфер для приема данных
**int count** – максимальное число элементов принимаемых данных
**int source** – процесс, от которого осуществляется прием сообщения
**int tag** – ожидаемый тэг сообщения
**MPI_Datatype datatype** – тип элемента принимаемых данных
**MPI_Comm comm** – коммуникатор
**MPI_Status *status** – статус завершения операции

*Рис. 1. Функции приема и отправки сообщений в MPI.*

Номер процесса, которому предназначено сообщение, передается в качестве параметра при вызове функции MPI_Send. Соответственно, номер процесса, от которого ожидается сообщение, указывается в качестве параметра функции MPI_Recv. Также обе функции указывают так называемый *тэг* сообщения. Прием сообщения происходит только в случае, если номер процесса, от которого ожидается прием, и ожидаемый тэг совпадают с номером пославшего процесса и тэгом посланного сообщения. Номера процессов имеют смысл по отношению к коммуникатору, который объединяет несколько процессов. Соответствующие операции посылки и приема сообщений должны выполнятся в пределах одного коммуникатора, который передается в качестве параметра при вызове функции MPI_Send и MPI_Recv.

При вызове функции MPI_Recv разрешается передавать в качестве номера и тэга значения параметров MPI_ANY_SOURCE и MPI_ANY_TAG соответственно. Эти значения называют *шаблонными*. Если в качестве номера процесса, от которого ожидается сообщение, указано шаблонное значение, то сообщение может быть принято от любого процесса из коммуникатора. Если в качестве параметра, задающего ожидаемый тэг сообщения, указан шаблон, то контроль тэга пришедшего сообщения не производится.

Сценарий работы параллельного приложения, состоящего из трех процессов, представленный на Рис. 2, демонстрирует возникновение недетерминизма в



случае приема по шаблону. Процессы *p* и *r* посылают процессу *q* сообщения, содержащие целые числа (1 и 2 соответственно). Процесс *q* дважды подряд выполняет прием сообщения и после каждого приема присваивает принятое значение переменной *a*. В зависимости от порядка, в котором осуществляется прием сообщений (варианты a и b на Рис. 2), значение переменной *a* после выполнение приема будет различным.

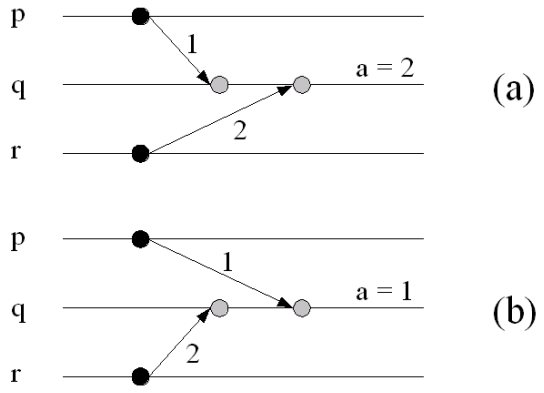


*Рис. 2. Недетерминизм, связанный с приемом по шаблону.*

Выбор процесса, сообщение от которого будет доставлено первым, может зависеть от относительной скорости работы процессов *p* и *r*. Эта ситуация называется *состоянием гонок* [3] (*race condition*). Состояние гонок является одной из основных причин недетерминизма в параллельных программах, исследованию этого эффекта посвящено большое число работ.

Недетерминированное поведение программы является фактором, существенно осложняющим процесс отладки и тестирования параллельных программ. Это связано главным образом с двумя причинами. Во-первых, ошибка, возникающая в процессе выполнения программы, может не воспроизводится под отладчиком, или воспроизводится не при каждом запуске. Во-вторых, при тестировании необходимо учитывать, что за счет недетерминированного поведения могут возникать ситуации, когда программа по-разному работает на тестовой и на реальной системе.

Таким образом, при тестировании и отладке параллельных программ возникает две проблемы, связанные с недетерминированным поведением:

- фиксация и воспроизведения хода выполнения параллельной программы;
- обход максимально-возможного числа путей выполнения параллельной программы.

## 2.3. Двойная типизация данных

Параллельные программы в средах MPI и PVM могут выполняться на вычислительном комплексе, различные узлы которого используют различные

форматы для представления данных. Вследствие этого, при передаче сообщений возникает потребность в обеспечении правильной интерпретации передаваемых данных на узле-получателе. Это достигается с помощью введения в библиотеку обмена сообщениями внутренних типов данных. Таким образом, появляется двойная типизация, при которой передаваемые данные обладают типом, определяемым языком программирования, и типом, определяемым библиотекой.

Двойная типизация может приводить к ошибкам двух видов. Ошибка первого вида содержится в следующем фрагменте кода программы:

```
int a=5;
MPI_Send(&a, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
```

Для передачи значения переменной целого типа используется библиотечный тип MPI_DOUBLE (должен быть MPI_INT). Это приводит к неправильной интерпретации библиотекой данных, хранящихся по адресу переменной **a**. Следствием такой ошибки может быть выход за границы адресного пространства процесса при попытке чтения байтов, расположенных вне памяти, выделенной под переменную **a**.

Второй характерный вид ошибок состоит в нарушении соответствия типов посылаемых и принимаемых данных. Пример нарушения такого соответствия приведен в следующем фрагменте кода:

```
if(rank == 0)
   MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
else
  MPI_Recv(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
```

Такое использование типов приводит либо к неправильным результатам, либо к ошибкам, происходящим в процессе выполнения программы.

Ошибки, причиной возникновения которых является нарушение правил работы с типами в параллельной программе, трудно выявлять как на этапе статического анализа, так и на этапе выполнения программы. Для облегчения обнаружения и диагностики таких ошибок требуется поддержка как со стороны системы времени выполнения, так и со стороны транслятора.

## *3. Теоретические концепции, используемые для анализа поведения параллельных программ*

Существует достаточно много моделей поведения параллельной программы. В данном разделе мы рассмотрим *событийную модель*, в которой процессы представлены последовательностью событий. Взаимодействия также моделируются как события определенного вида. Такие модели удобны для описания и для изучения свойств параллельных программ и часто используются в системах автоматизированного мониторинга, анализа и визуализации их поведения. Далее в разделе описывается сама модель и связанные с ней понятия логического времени и графа событий.



### 3.1. Логическое время

Основополагающей работой в этом направлении можно считать статью L. Lamport [5]. В этой работе вводится (вероятно, впервые) понятие *логического времени* в системе параллельных процессов, взаимодействующих при помощи отправки и приема сообщений.

Предполагается, что существует система параллельных взаимодействующих процессов. Каждый процесс рассматривается как последовательность событий. Событием является некоторое локальное действие в процессе, отправка или прием сообщения.

Для такой системы взаимодействующих процессов вводится отношение частичного порядка *случилось до*, обозначаемое через $\rightarrow$. Оно определяется как минимальное транзитивное отношение, удовлетворяющее следующим свойствам:

- если $a$ и $b$ – события одного процесса, и $a$ произошло до $b$, то $a \rightarrow b$;
- если $a$ - событие посылки сообщения, а $b$ - событие приема этого сообщения, то $a \rightarrow b$;
- если $a \rightarrow b$ и $b \rightarrow c$, то $a \rightarrow c$.

Обозначим первое из введенных отношений через $\xrightarrow{S}$, а второе – через $\xrightarrow{C}$. Тогда отношение $\rightarrow$ является транзитивным замыканием отношений $\xrightarrow{S}$ и $\xrightarrow{C}$.

Отношение "*случилось до*" позволяет определить так называемые *временные зоны* для события в параллельной программе. Для данного события $e$ определяются три временные зоны: PAST($e$), PARALLEL($e$) и FUTURE($e$), соответствующие группам событий, которые могут влиять на событие $e$, независимым от него событиям и событиям, на которые $e$ может оказывать влияние. Формально эти множества определяются следующим образом:

$$PAST(e) = \{f \mid f \rightarrow e\}$$
$$PARALLEL(e) = \{f \mid f \not\rightarrow e \wedge e \not\rightarrow f\}$$
$$FUTURE(e) = \{f \mid e \rightarrow f\}$$

*Логическим временем* для данного процесса $P_i$ называется функция $C_i$, отображающая события процесса $P_i$ во множество действительных чисел. Логическое время должно быть согласовано с отношением "случилось до", т.е., если $a \rightarrow b$, то логическое время события $a$ должно быть меньше логического времени события $b$.

Условие согласованности эквивалентно следующей паре условий:

- если $a$ и $b$ - события, принадлежащие одному процессу, и $a$ произошло до $b$, то $C(a) < C(b)$ ;

- если $a$ – событие отправки сообщения, а $b$ – событие приема этого сообщения, то $C(a) < C(b)$.

В работе [5] приводится одна из возможных реализаций логических часов для параллельных взаимодействующих процессов, которая удовлетворяет перечисленным условиям: каждый процесс $P_i$ снабжается счетчиком $C_i$, значение которого равно нулю в начале работы процессов и увеличивается в интервале между любыми двумя следующими друг за другом событиями.

При передаче сообщения процесс, посылающий сообщение, посылает вместе с ним также текущее значение логического время – временную метку (timestamp). При приеме такого сообщения принимающий процесс присваивает логическому времени значение не меньшее чем текущее значение времени в этом процессе и превосходящее значение, переданное с пришедшей временной меткой. Несложно видеть, что введенное таким образом логическое время удовлетворяет определению согласованности.

Недостатком такого определения логических часов является то, что они задают полный порядок на множестве всех событий параллельной системы: любые два события можно сравнить. При этом если $a$ и $b$ – события и $a \to b$, то $C(a) < C(b)$, но обратное не является верным. Например, в системе, состоящей из трех взаимодействующих процессов, изображенной на Рис. 3, $C(p_1) = 1 < 2 = C(q_2)$, хотя $p_1 \not\to q_2$.

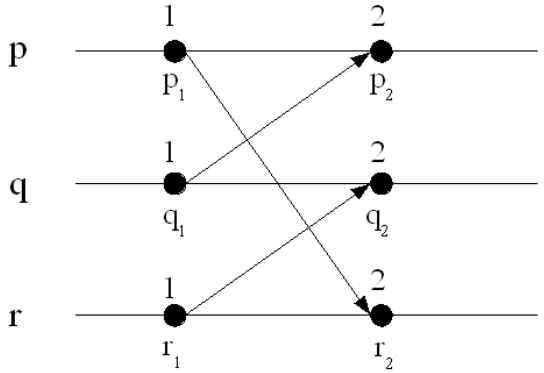


*Рис. 3.*

Таким образом, устанавливается искусственный порядок между событиями, которые на самом деле не являются упорядоченными.

Другой вариант логического времени, свободный от указанного недостатка, рассматривается в работе C. Fidge [6]. Рассмотрим вариант логического времени, предназначенный для системы процессов, в которой единственным способом взаимодействия является асинхронная отправка сообщения. Каждый процесс поддерживает *вектор счетчиков*, число компонент которого совпадает с числом параллельных процессов. Этот вектор изменяется по следующим правилам:



- Когда в процессе происходит любое событие он увеличивает значение компоненты вектора, соответствующей его номеру.
- Вместе с каждым посылаемым сообщением процесс посылает также текущий вектор счетчиков.
- При приеме сообщения процесс заменяет значение своего вектора счетчиков на покомпонентный максимум из его собственного и полученного вместе с сообщением векторов.

Несложно показать, что события $e$ и $f$, произошедшие на процессах $p$ и $q$ соответственно, удовлетворяют отношению "случиться до", тогда и только тогда, когда $t_e[p] < t_f[p]$ и $t_e[q] < t_f[q]$, где через $t_e[p]$ и $t_f[p]$ обозначены вектора счетчиков, связанные с событиями $e$ и $f$, соответственно.

Применим рассмотренную схему к системе взаимодействующих процессов, приведенной на Рис. 3. Результат представлен на Рис. 4. С событиями $p_1$ и $q_2$ связаны вектора [1, 0, 0] и [0, 2, 1] соответственно. Так как 1 > 0 и 0 < 2, то в соответствии со сформулированным критерием, события $p_1$ и $q_2$ не связаны отношением "случилось до".

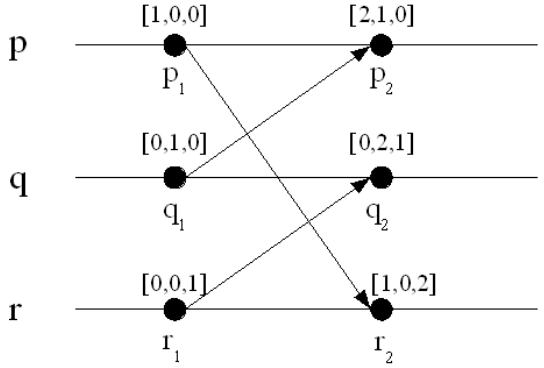


*Рис. 4. Логические часы Fidge.*

В [6] рассматривается модель взаимодействия процессов, усложненная за счет операций создания, завершения процессов и синхронизации процессов. В связи с этим модифицируются понятия отношения "случиться до" и по другому вводится логическое время.

Остановимся на синхронизации процессов. Под синхронизацией процессов $\langle p_{i_1}, \text{K}, p_{i_m} \rangle$ понимается совокупность событий $\langle e_{i_1}, \text{K}, e_{i_m} \rangle$ таких, что $e_{i_j} \in p_{i_j}$ для всех $j \in \overline{1,m}$, обладающих следующими двумя свойствами:

$$\exists k \in \overline{1,m}.(e \to e_{i_k}) \Rightarrow \forall j \in \overline{1,m}.(e \to e_{i_j})$$

$$\exists k \in \overline{1,m}.(e_{i_k} \to e) \Rightarrow \forall j \in \overline{1,m}.(e_{i_j} \to e)$$

Вектор счетчиков имеет один и тот же вид для всех событий, входящих во множество синхронизации, и вычисляется как покомпонентный максимум локальных векторов счетчиков для каждого из событий, входящих в это множество:

$$c := \max(c_{i_1}, \mathrm{K}, c_{i_m})$$

Автор также определяет правила вычисления вектора счетчиков для событий создания и завершения процессов и показывает, что частичный порядок, задаваемый введенным логическим временем, совпадает с порядком, определяемым отношением "случиться до".

## 3.2. Граф потока событий

Отношение «случилось до» задает частичный порядок на множестве событий, происходящих в параллельной программе. Представить этот порядок в виде ориентированного графа и применить его для анализа поведения параллельной программы предлагает D. Kranzlmüller в работе [7]. В этой работе определяется понятие *графа событий параллельной программы*. Граф событий представляет собой ориентированный граф, вершинами которого являются события, происходящие в параллельных процессах, а ребрами соединены события, находящиеся либо в отношении $\xrightarrow{S}$, либо в отношении $\xrightarrow{C}$. Другими словами, последовательные события в рамках одного процесса соединены ребрами, а также ребрами соединены операции посылки и соответствующего приема сообщения.

Рассматривается параллельная программа, состоящая из процессов, взаимодействующих при помощи посылки и приема сообщений. При этом рассмотрение ограничивается только случаем асинхронной посылки сообщений. Граф событий параллельной программы строится на основе информации, собираемой в процессе выполнения программы. Для этого исходная программа инструментируется вызовами функций, осуществляющих сбор и запись информации. Эти функции связываются с интересующими пользователя событиями и вызываются при их возникновении. В качестве примеров таких событий автор рассматривает отправку и прием сообщений и вызовы функций, возвращаемые значения которых не определяются входными данными программы, например функции генерации случайных чисел.

С помощью графа событий можно ввести понятие *эквивалентных выполнений* параллельной программы.

**Определение.** Два выполнения программы называются эквивалентными, если существует биективное отображение между вершинами графов событий, соответствующих обоим выполнениям, сохраняющее отношение "случилось до".

Построенный граф событий может быть применен в нескольких целях:

- для визуализации поведения программы;
- для воспроизведения записанного порядка событий при выполнении недетерминированной программы;



- для воспроизведения записанного порядка событий до установленной точки останова (breakpoint);
- для автоматизированной манипуляции порядком выполнения событий с целью проверки максимально-возможного числа вариантов поведения параллельной программы;
- для автоматизированного выявления аномалий поведения параллельной программы, являющихся указанием на потенциальную возможность ошибки;

Пример графа потока событий представлен на Рис. 5. Автор также указывает на возможность других способов визуализации потока событий.

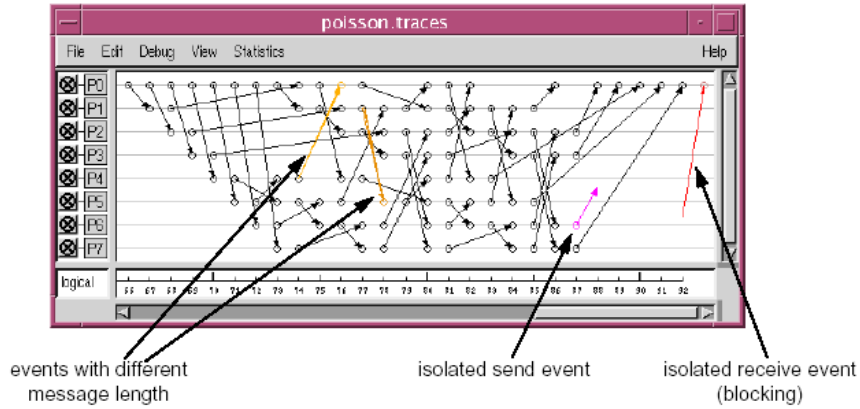


*Рис. 5. Граф событий параллельного приложения с указанием некоторых аномалий поведения: нарушения целостности сообщения, не доставленного сообщения и незавершенной операции приема. Иллюстрация взята из [7].*

## 4. Анализ программы во время выполнения

Целью анализа, проводимого на этапе выполнения программы является выявление ситуации, когда программа переходит в ошибочное состояние. Для параллельной программы такая ситуация может быть следствием того, что один из процессов перешел в ошибочное состояние, либо ошибочное состояние явилось следствием некорректного взаимодействия процессов. Таким образом, задача анализа того, что программа пришла в некорректное состояние в результате выполнения распадается на две подзадачи: выявление *локальных ошибочных состояний* (ошибочное состояние одного из процессов) и выявление *аномалий взаимодействия*. Рассмотрим подробнее каждую из этих подзадач.

## 4.1. Распознавание локальных ошибочных состояний

Локальные ошибочные состояния процесса параллельной программы можно разделить на две категории: к первой относятся все возможные виды ошибок последовательных программ, ко второй – ошибки, специфичные для используемой библиотеки или языка, обеспечивающего взаимодействия процессов. Распознаванию ошибок первой категории посвящено большое число работ, рассмотрение которых лежит за рамками данного обзора. Ряд работ посвящен проблеме выявления ошибок из второй категории.

В работе B. Krammer и др. [8] рассматривается ряд видов ошибок, характерных для программ, написанных на MPI. К этим ошибкам относятся: нарушение правил работы с коммуникаторами, определяемыми пользователем редукционными операциями и типами в параллельной программе. Для их выявления вызовы MPI-функций перехватываются, и нужная для анализа информация извлекается из аргументов, переданных при вызове, и сохраняется в соответствующих структурах данных в адресном пространстве процесса.

При обнаружении аномалии, например, попытки использовать в функции обмена освобожденный коммуникатор, процесс выдает диагностическое сообщение. В работе A. Claudio и др. [9] предлагается подход, в котором пользователю предоставляется возможность задать предикат, проверка истинности которого осуществляется процессом на этапе выполнения. При нарушении истинности выдается диагностическое сообщение.

## 4.2. Распознавание аномалий взаимодействия процессов

В случае параллельной программы, состоящей из процессов, взаимодействующих посредством передачи сообщений, наиболее распространенной аномалией взаимодействия является взаимная блокировка. Взаимная блокировка возникает в параллельной программе в ситуации, когда каждый процесс параллельного приложения не может выполнить какое-либо действие, связанное с взаимодействием с другим процессом, по причине неготовности последнего.

Для распознавания тупиковой ситуации в параллельной программе применяются главным образом два подхода: метод «тайм-аута» и метод анализа графа взаимодействия процессов. Метод обнаружения состояния блокировки по тайм-ауту [8, 10] состоит в том, что процесс, который не может выполнить команду передачи или приема данных в течение заданного порогового интервала времени, считается заблокированным. Если все процессы параллельного приложения заблокированы, то считается, что система находится в тупиковой ситуации и пользователю выдается соответствующее диагностическое сообщение. Достоинством этого метода является его универсальность: он применим к взаимодействиям любого типа и может быть использован в ситуациях, когда точный анализ состояния затруднен из-за неясной природы взаимодействия, например, в случае, когда в параллельной



программе, написанной на MPI, используются не предусмотренные MPI средства взаимодействия.

Недостатком метода тайм-аута является невозможность точного выбора порогового интервала, т.к. оно зависит от большого числа факторов (структура приложения, характеристики коммуникационной среды), влияние которых трудно учесть. При значении порогового интервала, близком к нулю, велика вероятность ложного распознавания тупиковой ситуации. При большом пороговом интервале существенным становится время задержки, необходимой для вынесения вердикта о наличии взаимной блокировки.

Преодолеть эти недостатки позволяет другой подход, предлагаемый J. Vetter и B. Supinski в работе [11]. Этот подход основан на анализе зависимостей между процессами параллельной программы. Граф зависимостей строится во время выполнения параллельной программы с помощью анализа информации о взаимодействиях. Специально выделенный *управляющий процесс* осуществляет мониторинг операций взаимодействия и помещает соответствующие маркеры в очереди событий, которые создаются и поддерживаются в его адресном пространстве для каждого из процессов параллельной программы.

Рекурсивная функция определения взаимоблокировки вызывается каждый раз, когда процесс, управляющий мониторингом, получает событие, соответствующее блокирующей операции обмена. Предположим, что пришедшее событие было помещено в очередь $q_A$, соответствующую процессу $A$. Из этой очереди выбирается самый «старый» маркер, соответствующий блокирующей операции. Предположим, что такая операция $a$ была найдена. Далее просматривается очередь сообщений $q_B$ процесса $B$, номер которого указан в качестве аргумента операции $a$. Если в очереди находится парная для $a$ операция $b$, перед которой нет блокирующих операций[1], то анализ прекращается, а операции $a$ и $b$ удаляются из очередей. Если в очереди нет парной операции и нет блокирующей операции, то анализ прекращается, т.к. недостаточно информации для того, чтобы сделать вывод о возможности завершения операции $a$. Если же в очереди $q_B$ есть блокирующая операция $b_1$, препятствующая выполнению парной операции, то для нее повторяется та же процедура, что и для операции $a$, а в граф зависимостей добавляется дуга, соединяющая процессы $A$ и $B$. Если в результате в графе зависимостей образуется цикл, то выносится вердикт о возникновении взаимной блокировки.

Данный метод позволяет выявлять также потенциально тупиковые ситуации, в которых взаимоблокировка может проявляться или нет в зависимости от состояния процесса вычислений и конфигурации системы. Зависимые от конфигурации и состояния процессов взаимоблокировки имеют место в системах, где функции для обмена сообщениями могут блокировать

[1] Две блокирующие операции в одной очереди могут появиться либо из-за несоответствия реальной картины результатам мониторинга, либо по причине использования операции, которые блокируются или нет в зависимости от размера сообщения MPI_Send.

выполнение процесса либо нет в зависимости от этих факторов. Примером такой системы является библиотека MPI, в которой то блокирует функция MPI_Send выполнение вызвавшего ее процесса или нет, зависит от размера сообщения, конфигурационных параметров и состояния системы. Рассмотрим ситуацию потенциальной взаимоблокировки, представленную на Рис. 6. Маркеры событий, находящиеся в очередях, образуют циклическую зависимость, которая указывает на то, что возможна взаимоблокировка. Вместе с этим, при наблюдаемом выполнении взаимоблокировки не произошло, о чем свидетельствуют события приема сообщений, расположенные в очередях после операций посылки.

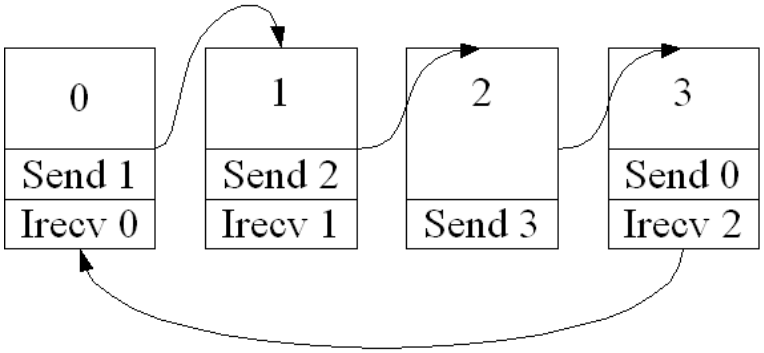


*Рис. 6. Пример циклической зависимости между блокирующими операциями. Пришедшая последней операция Send 3 замыкает цикл, и управляющий процесс выносит вердикт о наличии взаимоблокировки.*

## *5. Воспроизведение выполнения программы*

Параллельные программы обладают недетерминированным поведением, обусловленным различием в относительных скоростях работы процессов и каналов связи. Вследствие этого, программа может корректно выполняться на тестовой системе и завершаться с ошибкой на реальной системе. Для того, чтобы обеспечить воспроизведение на тестовой системе ошибки, проявляющейся на реальной системе, необходимы методы фиксации трассы выполнения программы и последующего воспроизведения программы в соответствии с записанной трассой. Рассмотрению этих методов посвящен данный раздел обзора.

### 5.1. Воспроизведение программы на основе графа потока событий

Граф потока событий может быть использован для воспроизведения некоторого пути работы недетерминированной параллельной программы. В процессе выполнения производится запись событий, происходящих в программе, создается т.н. *трасса*. При повторном запуске записанная



информация используется для того, чтобы повторить, каким образом реализовались события, соответствующие недетерминированным операциям в программе. Целью является получение выполнения программы, эквивалентного тому, для которого производилась запись трассы. В работе [7] рассматриваются три вида таких событий: прием по шаблону, неблокирующий прием и генератор случайных чисел.

Для приема по шаблону в трассу процесса, выполняющего прием, записывается номер процесса, пославшего сообщение. Для неблокирующего приема записывается, реализовался прием или нет. Для генератора случайных чисел в трассу производится запись возвращаемого значения.

На этапе воспроизведения прием по шаблону заменяется приемом от процесса с номером, записанным в трассу. Вместо неблокирующего приема выполняется пустой оператор столько раз, сколько раз прием не был реализован в соответствии с трассой, после чего выполняется блокирующий прием. Вызов датчика случайных чисел заменяется подстановкой значения, записанного в трассу.

Автор отмечает, что для целей воспроизведения достаточно записывать в трассу только события, соответствующие недетерминированным операциям в программе. Такой подход позволяет существенно сократить объем сохраняемой информации о работе программы.

В некоторых случаях требуется воспроизводить поведение не полностью, а до некоторой точки в параллельной программе, называемой *точкой останова* (*breakpoint*). Точка останова может быть задана как совокупность событий – по одному на каждом из процессов, ее достижение означает реализацию всех событий из этой совокупности и остановку сразу же после их выполнения. После остановки управление передается отладчику.

Проблема в выборе точки останова состоит в том, что выбранная произвольным образом совокупность событий может не быть достижима в процессе работы программы из-за ограничений, накладываемых передачей сообщений. D. Kranzlmüller предлагает в качестве решения использовать так называемые *остановочные срезы* (*breakpoint cuts*).

Остановочный срез строится по одному из событий $e$ на процессе $p$. На каждом процессе в это множество попадает наиболее позднее событие из числа находящихся в отношении "случилось до" по отношению к событию останова, т.е. из множества $PAST(e)$. Если множество событий на процессе $q$, находящихся в отношении «случилось до» по отношению к событию $e$ пусто, то точка останова на процессе $q$ полагается равной неопределенному значению *undefined*. Значение *undefined* означает, что состояние процесса в момент реализации точки останова неопределенно: это может быть последнее событие процесса или какое-то из событий, при котором дальнейшее выполнение невозможно из-за блокировки по пересылке данных. Остановочный срез, содержащий событие *undefined*, называется *частичным*. Частичный срез считается достигнутым, если произошли все определенные в нем события.

## 5.2. Частичное воспроизведение программы

Воспроизведение, начиная с первого оператора программы до точки останова, бывает достаточно большим для реальных приложений. Это может существенно осложнить процесс отладки. Поэтому представляется целесообразным подход, при котором в процессе работы приложения периодически записывается состояние программы, а программа воспроизводится, начиная от последнего записанного состояния до точки останова.

Состояние параллельной программы является совокупностью состояний составляющих ее процессов. Так как процессы взаимодействуют между собой, то не любая совокупность состояний процессов может рассматриваться как состояние параллельной программы. В частности, такая совокупность не может содержать события, находящиеся в отношении «случилось до». Таким образом, возникает две проблемы: как сохранять состояния процессов и как выбирать совокупность состояний для начала воспроизведения.

Работа [12] K. Mani Chandi и L. Lamport является одной из наиболее ранних работ, посвященных вопросам корректного сохранения состояния системы. В работе рассматривается совокупность параллельных процессов, взаимодействующих посредством обмена сообщениями по односторонним каналам связи, соединяющим процессы. Состояние системы определяется как совокупность состояний процессов и каналов. Процесс представляет собой последовательность состояний. Канал представляет собой очередь сообщений, в которую с одного конца передающий процесс помещает сообщения, а с другого конца принимающий процесс удаляет сообщения. Под состоянием канала понимается последовательность переданных в канал сообщений, которые еще не были приняты, т.е. находящихся в канале.

Поведение описанной системы моделируется как чередующаяся последовательность событий, каждое из которых изменяет состояние некоторого процесса и одного из инцидентных с ним каналов. Изменение состояния канала заключается в помещении в него очередного сообщения, если канал направлен «от процесса» или в удалении сообщения из канала, если он направлен «к процессу». Формально событие представляется набором $<p, s, s', M, c>$, где $p$ – номер процесса, в котором происходит событие, $s$ и $s'$ – состояния процесса до и после возникновения этого события соответственно, $M$ – сообщение, $c$ – канал по которому сообщение передается.

Событие называется *допустимым* в данном состоянии системы, если процесс $p$ находится в состоянии $s$, $c$ – канал, инцидентный процессу $p$ и, если $c$ – входной канал для процесса $p$, то $M$ находится в хвосте очереди (т.е. возможен прием этого сообщения процессом $p$). Если событие является допустимым, то в результате его выполнения система перейдет в новое состояние $S'$, где состояние процесса $p$ изменится на $s'$, а в канал $c$ будет помещено или удалено (в зависимости от направления канала) сообщение $M$. *Вычислением* называется любая допустимая последовательность событий, происходящих в системе. Некоторые системы допускают много различных вычислений.



Каждый из процессов может записывать свое состояние и состояние инцидентных с ним каналов. Задача заключается в том, как получить «осмысленное» состояние системы в результате такой записи. Проблема иллюстрируется примером системы, состоящей из двух процессов и некоторого объекта, называемого *токеном*, передаваемого от одного процесса другому.

Показывается, что если записи состояния различных процессов производить независимо в произвольных состояниях системы, то записанным может оказаться состояние, в котором в системе присутствует сразу два или не одного токена. Такие состояния не могут быть результатом вычисления и являются некорректными.

В работе предложен алгоритм, который позволяет записывать состояния, которые могут быть получены как промежуточное состояние системы в вычислении, полученным перестановкой событий вычисления, в процессе которого производилась запись. При этом последовательность событий и состояний обоих вычислений совпадают на участках до начала и после окончания записи. Содержательно, это свойство означает, что, во-первых, записанное состояние достижимо из начального, а конечное состояние достижимо из записанного.

Алгоритм, предложенный K. Mani Chandi и L. Lamport, состоит в следующем. После записи состояния процесс посылает маркер по всем инцидентным с ним каналам. Если процесс, получивший маркер, еще не производил запись своего состояния до момента получения маркера, то он записывает свое состояние и состояние канала $c$ как пустую последовательность. Если к моменту получения маркера процесс успел записать свое состояние, то состояние канала $c$ записывается как последовательность сообщений, полученных процессом по этому каналу с момента записи своего состояния до момента получения маркера. Таким образом, каждый процесс записывает свое состояние и состояние входящих каналов.

Остановка алгоритма за конечное время гарантируется только в ситуации, когда любой процесс в системе либо сам инициирует запись, либо существует путь в графе процессов, ведущий к нему из процесса, инициировавшего запись. Достижимость из записанного состояния конечного состояния является очень важным свойством и позволяет проверять *стабильные свойства*, т.е. свойства, верность которых в некотором состоянии влечет их верность в следующим за этим состоянием системы. Таким образом, если некоторое стабильное свойство верно в записанном состоянии, то оно верно и конечном состоянии. К таким свойствам, в частности, относится ситуация взаимоблокировки.

Другой подход к решению проблемы записи состояния параллельной программы для последующего воспроизведения предлагается в статье N. Thoai и др. [13]. В процессе работы параллельной программы производится запись состояний параллельной программы, называемых *контрольными точками*.

Контрольная точка параллельной программы определяется как совокупность контрольных точек составляющих ее процессов, называемых *локальными контрольными точками*. Сообщения, события посылки и приема которых

произошли до контрольной точки, не требуются при воспроизведении программы, начиная с этой точки. Остальные события можно разделить на две группы. К первой группе относятся сообщения, которые были посланы одним из процессов до контрольной точки, но не были приняты. Такие события называются *транзитными* (*in-transit*). Ко второй можно отнести сообщения, событие приема которых произошло до контрольной точки, а событие посылки произошло после контрольной точки. Такие события авторы статьи называют *сиротскими* (*orphan*).

Сообщение $m_1$ на Рис. 7 является примером сиротского, а $m_2$ – транзитного сообщения по отношению к контрольной точке $<C_{p1},C_{q2}>$.

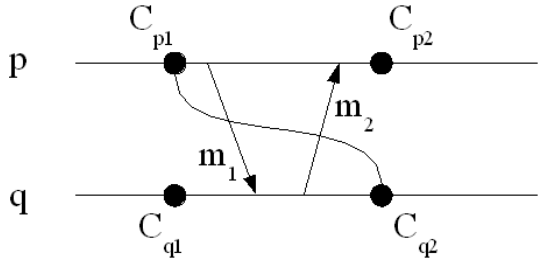


*Рис. 7. Сиротские и транзитные сообщения.*

Как было показано в работе [12] Chandi и др., не любая совокупность локальных контрольных точек может быть взята в качестве начальной для последующего воспроизведения программы. Простым следствием полученных в этой работе результатов является тот факт, что контрольная точка, в которой отсутствуют сиротские сообщения, может быть взята в качестве начальной для корректного воспроизведения программы. При этом все транзитные сообщения должны быть записаны и события их отправки должны быть воспроизведены при проигрывании программы с контрольной точки. Такие контрольные точки называются *целостными*.

Таким образом, если требуется начать выполнение программы с некоторой точки останова, то можно найти наименее удаленную от нее целостную контрольную точку и проиграть программу с нее до точки останова. Проблема заключается в том, что путь выполнения программы, который требуется воспроизвести, может быть достаточно длинным. Рассмотрим пример на Рис. 8. Ближайшей целостной контрольной точкой к точке останова $<B_p,B_q>$ является контрольная точка $<C_{p1}, C_{q1}>$, так как при любом другом выборе контрольной точки ближе к точке останова сообщение $m$ становится сиротским. При таком выборе контрольной точки время воспроизведения составит не менее четырех интервалов между контрольными точками.

Для решения данной проблемы N. Thoai и др. предлагают метод *пропуска сиротских сообщений*(*bypassing orphan messages*). Этот метод состоит в том, что информация о сиротских сообщениях сообщается процессам, и на этапе воспроизведения такие сообщения не посылаются. Например, в ситуации,



представленной на Рис. 8, можно начать выполнение программы с контрольной точки $<C_{p1},C_{q4}>$, которая существенно ближе к точке останова, чем целостная контрольная точка $<C_{p1},C_{q1}>$. При выполнении процесса $p$ отправка сообщения $m$ не производится, так как это сообщение является сиротским.

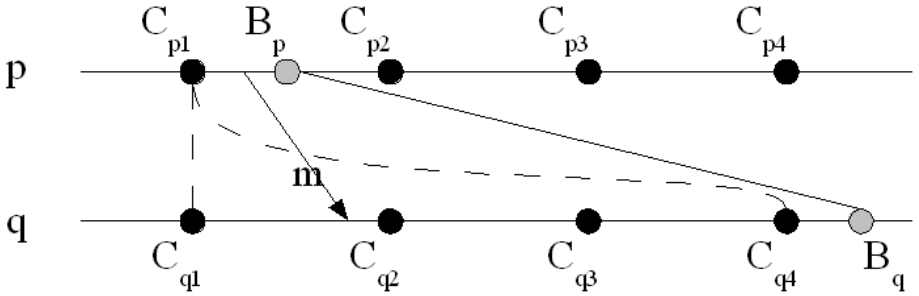


*Рис. 8. Целостные контрольные точки.*

Для выявления сиротских сообщений в рассматриваемой работе предлагается записывать в каждой контрольной точке номера последних принятых сообщений от каждого процесса. Предполагается, что все события отправки пронумерованы в порядке возрастания на каждом процессе. Вместе с сообщением посылается номер события отправки этого сообщения. При приеме сообщения, принятый вместе с ним номер сохраняется. При восстановлении с некоторой контрольной точки каждому процессу рассылается сохраненный номер последнего сообщения, полученного с него на другом процессе.

Далее при воспроизведении все события отправки сообщений процессу $q$ с номером, не превосходящим номер последнего сообщения, отправленного процессу $q$, считаются сиротскими и пропускаются.

Предлагаемая технология основана на наблюдении, что если событие отправки сообщения $m$ с процесса $p$ на процесс $q$ имело меньший номер, чем номер последнего принятого на процессе $q$ сообщения $m'$ от $p$, то процесс $q$ принял сообщение $m$. Это является верным наблюдением только в случае, когда между процессами $p$ и $q$ существует единственный FIFO канал, передающий сообщения. Для системы MPI это, вообще говоря, не является верным, поскольку если сообщения имеют разные тэги, то порядок их приема не определяется стандартом MPI.

Другая проблема, отмечаемая авторами, заключается в необходимости сохранять в трассе большое количество сообщений, которые потенциально могут стать транзитными. Если в трассе сохраняются все события отправки и получения сообщений, то работу программы можно воспроизводить, начиная с произвольной контрольной точки. Такой подход представляется достаточно неэкономным. Поэтому авторы предлагают альтернативный подход, предназначенный для ситуаций, когда не все события сохраняются. В этом случае, строится так называемый *R2-граф*. Вершины этого графа соответствуют локальным контрольным точкам, дуги соединяют

последовательные локальные контрольные точки. Также две контрольные точки $C_{p_i}$ и $C_{q_j}$,принадлежащие процессам $p$ и $q$ соответственно, соединяются дугой, если существует сообщение, отправленное в интервале между $C_{q\ i-1}$ и $C_{q_i}$, полученное в интервале между $C_{p_i}$ и $C_{p_{i-1}}$ (Рис. 9).

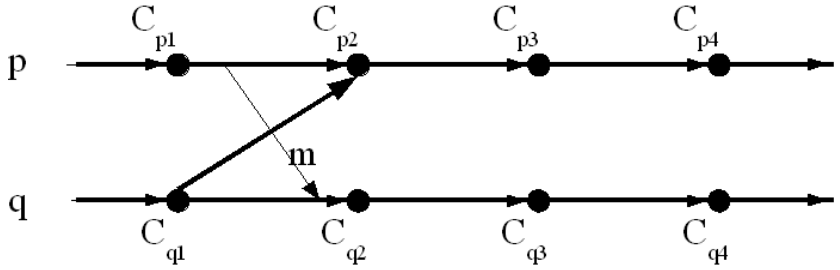


*Рис. 9. DR2-граф*

Путь в R2-графе, проходящий по вершинам в точности двух процессов, называется *DR2-путем*. В работе приводится формулировка и доказательство леммы, утверждающей, что если существует транзитное сообщение между процессами $p$ и $q$ для контрольной точки $\langle \mathrm{K}, C_{p_i}, \mathrm{K}, C_{q_j}, \mathrm{K} \rangle$, то существует *DR2-путь*, соединяющий $C_{p_i}$ и $C_{q_j}$. Из этого утверждения следует, что контрольная точка, любые две вершины которой не соединены DR2-путем, может быть линией восстановления, так как все транзитные сообщения записаны в трассу.

Возможны различные стратегии, определяющие какие события требуется сохранять в трассу. В работе [13] конкретные стратегии не рассматриваются. Одна из возможных стратегий рассматривается в [14]. Предлагаемая методика Rollback-one-step checkpointing (ROS) позволяет существенно сократить объем сохраняемых событий, при этом, максимальная возможная дистанция между точкой останова и ближайшей линией восстановления не превосходит двух контрольных интервалов.

## 6. Манипуляция поведением недетерминированной программы

Параллельные программы могут быть недетерминированными. При повторных запусках программы под управлением отладчика или на тестовой системе может реализовываться только часть возможных путей выполнения параллельной программы, в то время как при запуске на целевой системе выполнение может пойти по другому пути, приводящему к ошибке. Поэтому на этапе отладки и тестирования целесообразно проверить корректность работы программы на максимально большом числе возможных путей выполнения. Для этого применяется *манипуляция событиями* параллельной программы, которая заключается в том, что при воспроизведении программы изменяется порядок



событий, в случае, если он определен неоднозначно. Например, изменяется ранг процесса, от которого осуществляется прием по шаблону.

В [7] предлагается ручная и автоматизированная манипуляция событиями. Рассматриваются три вида событий, которые порождают недетерминированное поведение, и, соответственно, могут быть объектами манипуляции: вызов генератора случайных чисел, неблокирующий прием и прием по шаблону. Общая идея манипуляции состоит в том, что в записанном графе событий изменяется реализация приема сообщений по шаблону, граф событий модифицируется и программа воспроизводится на основе модифицированного графа событий. При этом воспроизводить поведение программы можно только до тех пор, пока ее поведение не зависит от результатов манипуляции. Для того, чтобы разделить события, не зависящие от манипуляции событием *e*, от всех прочих применяется *перестановочный срез*, который состоит из первых событий множества *FUTURE(e)*.

Манипуляция производится следующим образом:

- Выбирается событие для манипуляции.
- Для выбранного события строится перестановочный срез.
- Каждый из процессов параллельного приложения воспроизводится в соответствии с записанным порядком событий до события, входящего в перестановочный срез.
- Процесс, содержащий событие для манипуляции выполняется в соответствии с записанной трассой до этого события, а само событие выполняет в соответствии с результатом манипуляции.
- После перестановочного среза все процессы приложения выполняются без ограничений, и записывается новая трасса.

Манипуляция событиями может осуществляться как вручную, так и автоматически. Для генератора случайных чисел она состоит в том, чтобы выдать значение датчика, отличное от записанного в трассу. Для неблокирующего приема завершившегося успешно – на замене его на прием, завершившийся неуспешно, т.е. не принявший сообщение, или наоборот. Для приема по шаблону реализовавшийся прием заменяется на прием от другого кандидата. При ручной манипуляции пользователь самостоятельно выбирает другую реализацию из числа возможных на основе визуального представления.

При автоматической манипуляции этот выбор осуществляется автоматически. Например, в случае приема по шаблону по графу событий определяются возможные события отправки сообщений, от которых мог произойти прием. Затем выбирается одно из этих событий. После чего выполнение программы воспроизводится с учетом манипуляции. При этом программа выполняется по-другому, в результате получается другой граф событий, возможно не эквивалентный исходному. Таким образом, можно говорить о дереве выполнений программы, в котором вершины соответствуют выполнениям программы, а ребра соединяют выполнения, полученные друг из друга одной манипуляцией событиями. Если в этом дереве объединить вершины, соответствующие эквивалентному выполнению, то получиться граф общего

вида. Отождествление эквивалентных выполнений позволяет существенно сократить перебор вариантов.

В [7] признается, что непосредственное применение предложенной методики для тестирования недетерминированного поведения параллельной программы может быть затруднительным из-за необходимости перебора большого числа вариантов. В качестве решения этой проблемы предлагается три подхода: использование информации от пользователя, применение статического анализа и оценка вероятности появления тех или иных событий, производимая на основе информации об архитектуре параллельной платформы.

Другой подход к систематической отладке параллельных программ предлагается Kacsuk и др. в работах [15, 16]. Рассматривается система отладки параллельных программ DIWIDE (Distributed Windows Debugger), которая интегрирована с системами PGRADE и WINPAR. Отладка в DIWIDE построена на понятиях коллективной точки останова и макрошага. *Коллективной точкой останова* называется совокупность точек останова процессов, где в каждом из процессов останов поставлен на функции обмена. *Макрошагом* называется совокупность последовательностей команд процессов между следующими друг за другом точками останова. Рассмотрение ограничивается только т.н. *чистыми макрошагами*, которые не содержат инструкций обмена внутри участков, образующих макрошаг.

В результате выполнения шага каждая из локальных точек останова может быть в *активном* или в *спящем* состоянии. Активное состояние возникает в случае, если макрошаг выполнен и процесс остановился перед данной точкой останова. Спящее состояние является индикацией того, что соответствующая коммуникационная операция не завершилась по тем или иным причинам. Проблема определения завершения шага решается в DIWIDE с помощью time-out. Если по прошествию time-out все спящие точки останова продолжают оставаться такими, то шаг считается завершенным и отладчик прекращает его выполнение.

DIWIDE поддерживает несколько возможных способов отладки параллельных программ. При пошаговом выполнении отладчик последовательно производит макрошаг, завершает его, пользователь или сам отладчик выбирает следующую точку останова. После чего снова производится макрошаг. Следующая точка останова выбирается так: для каждого процесса, на котором есть активная точка останова, вычисляется следующая инструкция пересылки, на ней устанавливается следующая точка останова. Если в программе присутствует оператор, меняющий поток управления, то точка останова устанавливается на инструкциях обмена в каждой из возможных ветвей выполнения.

При систематической отладке программа полностью выполняется, при этом создается трасса, в которой фиксируется порядок выполнения приемов по шаблону и ветви операторов ветвления. Затем производится повторное выполнение программы до последней точки, в которой встречается оператор ветвления или прием по шаблону. Далее пользователю предлагается выбрать ветвь оператора ветвления, по которой далее следует продолжить выполнение



программы или от какого процесса производится прием по шаблону. Выбор может также производиться автоматически отладчиком. Повторное выполнение это процесса позволяет осуществлять систематический обход дерева возможных состояний программы.

Возможность систематического обхода дерева состояний параллельной программы в DIWIDE была использована для автоматической проверки некоторых свойств параллельной программы, записанных в виде формул временной логики [17], описана в работе J. Kovacs и др. [18]. Примером такого свойства может быть утверждение: "если данные были помещены в буфер, то при любом возможном пути выполнения программы наступит состояние, когда данные будут из него извлечены".

Для выражения формул временной логики применяется разработанный для этих целей API на языке Java. Проверка истинности формулы производится стандартным методом с помощью систематического обхода дерева состояний и проверки истинности логических утверждений в каждом из узлов. Обход управляется специальным модулем, который использует API, предоставляемый отладчиком DIWIDE для обхода состояний. В каждом состоянии проверяется истинность предикатов, которые зависят от атомарных формул. Атомарные формулы представляют собой функции, работающие на стороне приложения и возвращающие информацию о текущем состоянии приложения. Эти функции реализует пользователь. В результате проверки, выполняемой в процессе обхода, выносится вердикт об истинности формулы для данного параллельного приложения.

## *7. Инструментальные средства*

К настоящему моменту разработано большое количество программных средств для мониторинга, визуализации и анализа поведения параллельных программ. В данном разделе рассматриваются форматы хранения трассировочной информации (пункт 7.1), инструменты для ее сбора (пункт 7.2), средства визуализации трасс и их автоматизированного анализа (пункт 7.3), средства манипуляции поведением параллельной программы (пункт 7.4) и интегрированные среды для отладки параллельных приложений (пункт 7.5).

### 7.1. Стандартные форматы записи трассы

**ALOG, CLOG, SLOG**

Профилировочная библиотека MPE (Multi-Processing Environment) [19, 20], поставляемая вместе с пакетом MPICH, может писать трассы в трех форматах – ALOG, CLOG и SLOG. Формат ALOG устарел и поддерживается только для совместимости со старыми программами. Он осуществляет запись событий в трассу как текст формата ASCII.

Формат CLOG похож на ALOG, но помещает данные в двоичном формате. Трасса представляет собой последовательную совокупность событий, определенных пользователем, с временными отметками.

Формат SLOG (Scalable Log) является наиболее мощным форматом при профилировании. При его использовании трасса пишется в специальном двоичном формате, который позволяет программе, осуществляющей визуализацию, работать с большими лог-файлами.

Каждый из этих форматов трассы связан с программой для ее визуализации. Для визуализации трассы в формате ALOG используется nupshot, для трасс в формате CLOG применяются nupshot и jumpshot, для SLOG – Jumpshot-3.

**PICL**

В библиотеках PICL (Portable Instrumented Communication Library) [21, 22] и MPICL, описанных ниже, предусмотрена возможность записи трассы параллельной программы. Каждое событие в трассе представляет собой запись, содержащую следующие поля:

- `record type` – тип записываемой информации;
- `event type` – тип события, с которым связана записываемая информация;
- `timestamp` – время события;
- `processor id` – номер процессора;
- `process id` – номер процесса;
- `number of data fields` – число других полей данных, связанных с событием;
- `data descriptor` – формат дополнительных данных;
- `data` – другие данные.

Существует соглашение о том, данными каких типов должны быть заполнены перечисленные поля, но это не стандарт. Таким образом, пользователь сам может решить, какие данные и в каком формате требуется писать в трассу. Для этого он должен в начало файла трассы включить запись о том, как читать и как интерпретировать эти добавленные поля. Трасса представлена в ASCII, поэтому может просматриваться пользователем без специальных инструментов.

**EPILOG**

Выполняемая программа с вызовами библиотеки времени выполнения EPILOG (Event Processing, Investigating and Logging) [23] генерирует трассу в одноименном формате EPILOG. Трасса состоит из записей определений объектов и записей событий с временными отметками. Записи событий описывают поведение программы и определенных объектов. В трассу помещается вся необходимая информация о командах программы и вызовах процедур, метриках времени выполнения(например, program counter), и коллективных процедурах.

Файл трассы формата EPILOG последовательно содержит в себе:

- заголовок;
- записи определений;
- записи событий.



В заголовке последовательно содержится строка "EPILOG", символ нуля, номер версии EPILOG, и байт, означающий байтовый порядок для арифметических операций.

Записи определений объектов нужны для уменьшения информации, помещаемой при записи событий, связанных с описанными объектами. Такими объектами могут быть участки кода программы, имена файлов и т.д. Каждая запись состоит из заголовка и тела записи. Заголовок записи состоит из двух байтов. Первый байт содержит длину тела записи, второй байт – ее тип. Затем следует само тело записи.

Файл трассы формата EPILOG бинарный, поэтому для облегчения доступа к информации, содержащейся в нем, была разработана библиотека EARL [24, 25, 26, 27, 28] – Event Analysis and Recognition Library (Language – название варьируется в литературе), высокоуровневый интерфейс для доступа и вычисления информации о трассах в формате EPILOG.

EARL предлагает следующую функциональность:

- доступ к произвольному событию
- доступ к состоянию программы в определенное время
- ссылки на связанные события
- различные статистические функции

Центральной абстракцией в EARL является событие. Каждое событие должно иметь тип, время и место. Для некоторых типов событий есть специфические атрибуты, включающие ссылки на связанные объекты. EARL предлагает абстрактные типы событий, которые на самом деле не записаны в трассу, но могут быть вычислены по ней. Библиотека EARL предоставляет множество функций (C++ API и Python API) для доступа к этим данным, содержащимся в трассе формата EPILOG. На саму трассу накладывается несколько ограничений, которые предусмотрены форматом EPILOG, например, что посылка сообщения должна встречаться в трассе раньше, чем его прием.

**SDDF**

(Self-Defining Data Format) [29, 30] – гибкий мета формат записи трассы, который предусматривает запись и данных, и их структуру. Формат не описывает заранее фиксированное множество типов, которые могут быть использованы, или как эти типы должны быть записаны. Формат также не диктует семантику записываемых данных. Корректный SDDF файл – тот, который следует синтаксису мета формата SDDF. В целях увеличения компактности и переносимости было разработано две SDDF грамматики – для двоичного представления и для ASCII. Соответственно, двоичное представление более компактно, а представление в ASCII – переносимо. Были разработаны и утилиты конвертирования этих представлений друг в друга.

В дополнение к мета формату была разработана библиотека классов C++, обеспечивающая интерфейс данных, представленных в SDDF файле. Она поддерживает полный набор операций с данными, необходимыми как для записи SDDF файлов, так и для интерпретации данных, записанных в SDDF файле.

Мета-формат SDDF предусматривает четыре типа записей:

- команды;
- информация о файле;
- описатель структуры записи;
- данные записи.

Соответственно, SDDF файл имеет следующую структуру:

- заголовок – запись типа информация о файле;
- описатели структур записей;
- записи данных.

В заголовке содержится информация о типе SDDF файла (двоичный или ASCII), время и дата его создания, и другая необходимая информация. В описателе структуры записи содержится номер описателя, имя и типизированные атрибуты. В записи данных содержится имя (для ASCII, для двоичного формата – номер) и значения атрибутов, указанных в соответствующем описателе. Типичным атрибутом записи является время (timestamp).

Записи-команды могут быть помещены между другими записями и обеспечивают связь между различными процессами, использующими SDDF файл. Они могут быть добавлены приложением или самим пользователем для передачи необходимой информации соответствующему анализирующему модулю.

**STF** (Structured Trace file Format) и **VTF** (VampirTrace Format)} – форматы трасс, используемые Intel Trace Collector [31] и VampirTrace [32, 33, 34]. Формат STF подразумевает запись трассы как в один, так и несколько файлов с целью преодоления возможных ограничений размеров файлов. Трасса состоит из нескольких фреймов. Если трасса находится в одном файле – то все они в нем, если в нескольких – то каждый фрейм в своем файле. Разделение трассы на фреймы происходит по трем принципам:

- по времени;
- по процессам;
- по типы данных трассы.

Формату трассы STF предшествовал формат VTF (Vampir Trace Format). STF превосходит VTF по нескольким параметрам – каждый файл содержит только специфические для него данные, доступ к которым проще и быстрей (и для этого не требуется просматривать все файлы), формат трассы более расширяем и переносим, более прозрачен и структурирован, и он наиболее компактен. Кроме того, в этом формате подразумевается параллелизм по записи и чтению данных в трассе. Изначально, трасса в формате STF представлена следующими файлами:

- один индексный файл `(<trace>.stf)`
- один файл с декларативными записями `(<trace>.stf.dcl)`
- один файл с фреймами `(<trace>.stf.frm)`
- один статистический файл `(<trace>.stf.sts)`
- один файл сообщений `(<trace>.stf.msg)`
- один файл глобальных операций `(<trace>.stf.gop)`



- один или несколько файлов процессов `(<trace>.stf.pr.<index>)`
- для трех вышеупомянутых файлов один связующий файл (каждый с расширением `.anc`)

Формат трассы STF – двоичный. Специальная утилита stftool может читать трассы в формате STF. Кроме извлечения данных, эта утилита позволяет конвертировать трассу в более ранний формат – VTF. Формат VTF предусмотрен в двух вариантах – как в двоичном представлении, так и в ASCII (соответственно, двоичный более компактен, ASCII – может читаться пользователем без специальных инструментов).

**XMPI**

XMPI [35] – формат трасс, записанных с помощью одноименного монитора XMPI. Большинство сущностей в трассе соответствует структурам C, и описываются в терминах этих структур. Формат трассы состоит из последовательности трасс XMPI. Трассы XMPI делятся на три категории

- `world traces` – трассы, содержащие "магические" числа и описание процессов
- `object traces` – трассы, описывающие MPI-объекты такие как типы данных и коммуникаторы
- `run-time traces` – трассы, описывающие события времени выполнения такие как посылка/прием

Трассы, описывающие события времени выполнения для каждого процесса следуют в хронологическом порядке внутри файла трассы. Для разных процессов этот порядок необязателен.

Для многих перечисленных форматов трасс существуют утилиты конвертирования из одного формата в другой. Например, в составе инструмента KOJAK, работающего с трассами формата EPILOG с помощью EARL, есть утилита конвертирования трассы из EPILOG в VTF-ASCII. В составе пакета Pablo есть утилита конвертирования трасс в формате VTF, PICL и AIMS (не описанный здесь) в формат SDDF. В MPE входят утилиты конвертирования форматов ALOG, CLOG и SLOG друг в друга.

## 7.2. Библиотеки для сбора трасс параллельных приложений

**PICL и MPICL**

Библиотека PICL (Portable Instrumented Communication Library) [21] была разработана в Национальной Лаборатории в Ок-Ридже в 1989. Данная библиотека предназначена для обеспечения взаимодействия параллельных процессов через посылку сообщений. В библиотеке предусмотрена возможность записи трассы в формате PICL.

Все события сохраняются в текстовом формате, что облегчает их восприятие пользователем. В дальнейшем, с появлением стандарта MPI, возможности трассировки, предусмотренные в PICL, были перенесены на MPI, в результате

чего появилась библиотека MPICL. Для поддержки с MPI в MPICL предусмотрены следующие возможности:

- Для представления команд MPI, не имеющих аналогов в PICL, добавлены новые типы событий. При этом базовый формат трассы сохранен.
- Все команды инструментирования MPICL вызываются через интерфейс MPI_PCONTROL. Таким образом, программа может выполняться как вместе с MPICL библиотекой, так и без нее – в зависимости от того, собрано приложение с библиотекой или нет.

**MPE**

MPE (Multi-Processing Environment) [19] включает несколько библиотек для сборки информации о выполнении MPI-программ, включая лог-файлы для посмертного отображения и анимации времени выполнения. Основными процедурами MPE являются:

- процедуры, обеспечивающие доступ к разделяемому дисплею;
- сборщики трасс (во время работы программы каждый процесс пишет свой собственный лог-файл, после окончания работы все лог файлы собираются вместе для общего представления и обработки рассогласования часов);
- процедуры последовательных секций – для упорядочения выполнения одного кода различными процессами;
- обработчики ошибок позволяют пользователю управлять реакцией на ошибки времени выполнения.

Лог-файлы могут быть созданы как с помощью прямого инструментирования кода вызовами процедур MPE, так и при помощи связывания программы с соответствующими MPE библиотеками. Существуют три профилирующие библиотеки:

- `tracing library,` трассируются все вызовы MPI с добавлением предварительной строки с указанием номера процесса, и заключительной строки (указывающей, что вызов завершился);
- `animation Library` – библиотека анимации времени выполнения программы;
- `logging library` – могут писать лог файлы в 3 разных форматах – CLOG, ALOG, SLOG(по умолчанию используется CLOG).

Для того чтобы создать лог-файл программа должна вызвать соответствующую процедуру, затем каждый процесс должен вызвать другую процедуру, и в конце также все процессы должны вызвать процедуру, которая объединяет все лог файлы вместе.

Для измерения продолжительности какого-либо участка программы (называемого состоянием), необходимо, чтобы события его начала и конца



были отмечены вызовами специальных процедур. И так для каждого интересующего участка программы.

**EPILOG**

В набор инструментов KOJAK [36] входит EPILOG (Event Processing, Investigating and Logging) – библиотека создания трасс времени выполнения. Исходная программа инструментируется вручную либо автоматически (инструментируются все вызовы MPI и предопределенные пользователем вызовы). Все инструментированные вызовы вызывают функции библиотеки EPILOG, которая обеспечивает механизм буферизации и способ создания трасс выполнения программы. В конце инструментирования получается инструментированная выполняемая программа. Выполнение этой программы производит трассу в формате EPILOG, к которой в дальнейшем можно обращаться с помощью EARL.

**VampirTrace и Intel Trace Collector**

Инструмент Intel Trace Collector [31] раньше был известен как VampirTrace [34]. Фактически, это инструментированная MPI библиотека. Процесс инструментирования программы заключается в ее связывании с библиотекой VampirTrace, которая фактически является профилирующей прослойкой между программой и MPI библиотекой. Выполнение этой программы производит трассу в формате STF или VTF по выбору пользователя (ранее только VTF). Эти и другие настройки описываются в специальном конфигурационном файле.

В трассу записываются все вызовы MPI-библиотеки, а также предопределенные пользователем события. Пользователю также позволяется приостанавливать и начинать снова процесс трассировки. Это делается с помощью специального VampirTrace API. С помощью этого API и конфигурационного файла позволяется настраивать фильтрацию записываемых событий по заданным правилам.

**Pablo Trace Library**

Pablo [30] состоит из нескольких компонентов инструментирования, трассировки и анализа параллельной программы. Для трассировки MPI событий используется профилирующий интерфейс, поэтому требуется только связать программу с расширенной трассировочной библиотекой. Кроме трассировки этих событий, позволяется писать в трассу вызовы процедур, входные/выходные данные, а также определенные пользователем события. Для последнего требуется соответствующее ручное инструментирование вызовами соответствующих функций библиотеки Pablo Trace Library. После этого программа выполняется обычным способом. Каждый процесс пишет свой трассировочный файл, и эти файлы в дальнейшем могут быть объединены в один. В процессе выполнения также позволяется останавливать и возобновлять процесс трассировки программно.

## 7.3. Средства визуализации и автоматизированного анализа выполнения параллельной программы

### ParaGraph

Система ParaGraph [37] предназначена для анимации трасс, записанных в формате PICL. Поддерживается несколько различных визуальных представлений выполнения программы. Показывается загрузка процессоров во время выполнения, трафик в сети, очереди сообщения и другая информация, полезная для понимания работы программы.

Для анимации используются несколько различных представлений – по выбору пользователя (Рис. 10). Узлы сети (процессоры) отображаются кругами разного цвета в зависимости от статуса процесса (занят, простаивает, посылает, получает). Дуги, соединяющие круги, отображают посылку и прием сообщений. Изображение меняется динамически в процессе анимации.

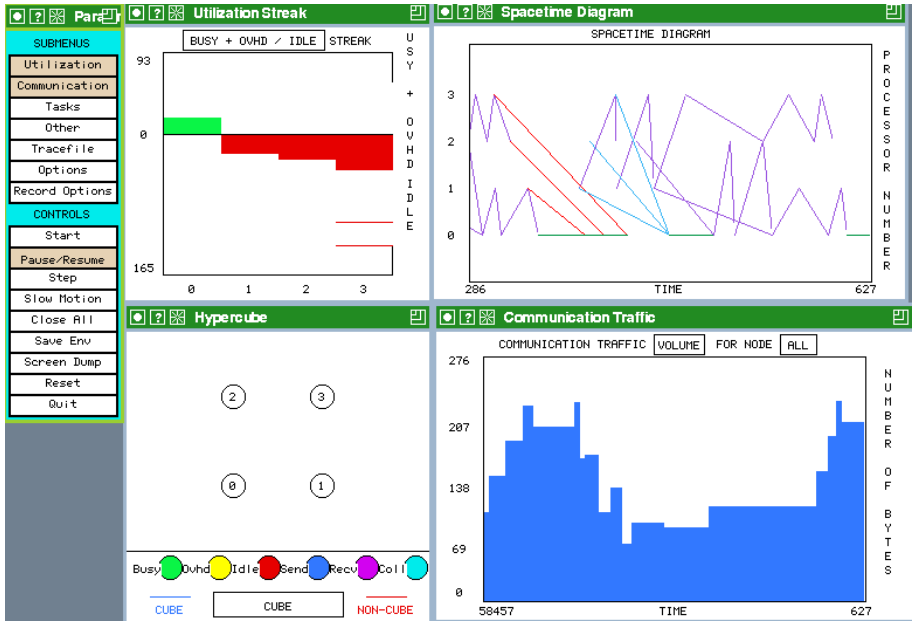


*Рис. 10. Главное окно ParaGraph*

Список отображаемых процессов, отображаемые участки трассы, степень детализации информации могут меняться пользователем в процессе анимации. Это позволяет изучать с разной степенью детализации различные участки трассы, концентрируя внимание на объектах, представляющих наибольший интерес.

Для отображения событий процессов используется временная диаграмма (time-space diagram). На шкале времени может отображаться как логическое (используются часы Lamport), так и физическое время. Взаимодействия процессов отображаются с помощью дуг, соединяющих события на различных



процессах. При этом цвет дуги соответствует степени завершенности взаимодействия: сообщение послано, находится в очереди, доставлено.

### ATEMPT

Инструмент ATEMPT (A TOOL for EVENT MANIPULATION) представляет возможности для визуализации выполнения параллельной программы в виде временной диаграммы. ATEMPT – основная составляющая другого инструмента – MAD [38, 39], отвечающая за графическое представление. В качестве исходных данных для визуализации используется трассировочная информация, собранная с помощью инструмента NOPE [40, 41]. В процессе визуализации трассы ATEMPT автоматически определяет такие ошибки, возникающие при взаимодействии процесс, как несовпадение длин посланного и принятого сообщения, не принятое сообщение или блокировка процесса на операции приема. Пользователю предоставляется доступ к различной информации, связанной с событием, в частности, возможно указание строки в исходном тексте программы, в которой произошло данное событие. Для этого файл с исходным текстом открывается в окне редактора и строка, соответствующая событию, выделяется цветом.

Так как ATEMPT предназначен для интеграции с другими инструментами, и в частности с инструментами, предназначенными для повторного воспроизведения программы, то в нем предусмотрены возможности для выявления условия гонок и перестановки последовательности, в которой осуществляется прием. Также предусмотрена возможность установки точек останова, в соответствии с теоретическими разработками D. Kranzlmüller [7], рассмотренными в разделе 6 данного обзора: пользователь выбирает событие на одном процессе, а точки останова для других процессов устанавливаются автоматически.

### POET

Poet (Partially Ordered Event Tracer) [42, 43, 44, 45, 46] – инструмент для сбора и представления трасс событий выполнения параллельных программ. POET позволяет программе работать как в нормальном режиме, как в режиме повторного выполнения (replay), так и в режиме посмертного анализа(post mortem).

Основное внимание POET уделяет проблемам масштабируемости и сложности представления трасс выполнения программ. Важным достижением здесь является применение абстракции в области процессов и в области событий таким образом, что пользователю предоставляется выполнение программы в терминах более высокого уровня, чем процессы и единичные события. Однако один единственный уровень абстракции не приемлем для большинства программ. Поэтому POET предлагает иерархическое представление, которое позволяет наблюдать и оценивать работу программ на разных уровнях абстракции. На всех уровнях абстракции пользователю предоставляется набор трасс (в виде горизонтальных линий), где каждая трасса состоит из последовательности событий, представленных различными символами. Расположение событий на трассах говорит о временных связях между ними. Эти связи могут быть основаны как на физическом времени случившихся событий, так и на логическом времени.

Трассы соответствуют либо одному процессу, либо кластеру. Кластеры формируются объединением вместе нескольких процессов. Кластеры представляются одной или несколькими внешними трассами в зависимости от того, случившееся внешнее событие полностью или частично упорядочено в кластере. Внешние события – те, которые включают взаимодействие с процессом за пределами данного кластера, а внутренние – которые происходят внутри кластера. События могут быть разделены на примитивные и абстрактные. Примитивные – события нижнего уровня абстракции. Абстрактные включают в себя несколько примитивных событий.

Кластеры и процессы отличаются цветом при отображении на экране. Примитивные события отображаются пустыми и заполненными кружками и прямоугольниками. Абстрактные события, затрагивающие несколько трасс, представляются прямоугольниками, пересекающими те трассы, которые содержат примитивные события, соответствующие этому абстрактному событию. Типичный пример приведен на Рис. 11.

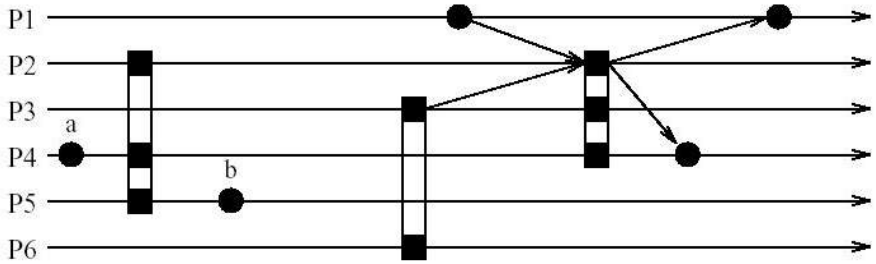


*Рис. 11. Представление POET абстрактных и примитивных событий*

В больших программах для абстракции требуется автоматическая поддержка, как для процессов, так и для событий. Но даже в том случае, когда эта поддержка реализована, может быть, множество трасс и событий, которые не помещаются на одном экране. Для этого требуется возможность прокрутки, как по трассам, так и по времени.

Одна из проблем заключается в том, что невозможно представить всю историю выполнения программы на одном экране. Для большинства трасс подходит прокрутка, но для большого количества событий – это неудобное решение. Прокрутка требует соответствующей зависимости между позицией указателя и соответствующими данными. Для физического времени такая зависимость существует. Для частично упорядоченных событий требуется альтернатива.

Принятая техника основывается на выборе события и задания его либо самым левым, либо самым правым событием при отображении. Затем строится отображение событий, которое соответствует частичному порядку при условии, что события из предыдущего отображения удаляются, только если это необходимо. Основная идея в том, что вместо отображения событий на фиксированных позициях по отношению друг к другу, пользователь должен



представить, что события – это шарики, нанизанные на провода, и некоторые провода соединены между собой. Прокрутка по истории событий включает скольжение по этим шарикам на проводах с перемещением других шариков, если это необходимо. Подробно механизмы, применяемые для организации прокрутки описаны в [47].

POET имеет модульную архитектуру (Рис. 12).

Stands for Partial-Order Event Tracer

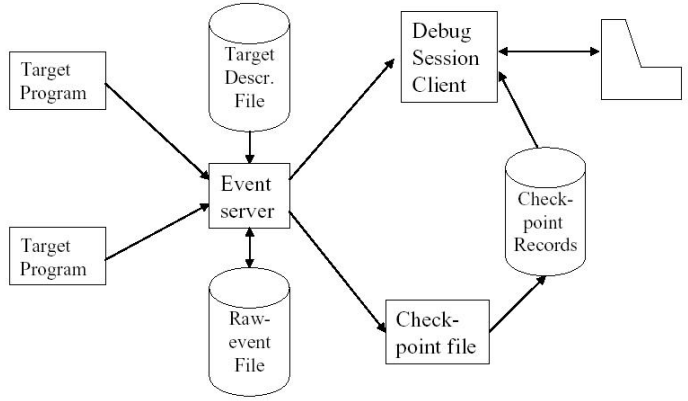


*Рис. 12. Архитектура POET*

POET включает в себя как минимум 3 процесса. Первый из них – disk server – взаимодействует с программой пользователя. Он получает информацию о событиях от инструментированной программы и записывает ее в трассу. Этот процесс посылает информацию о событии клиентам в ответ на соответствующий запрос.

Пользователь взаимодействует с POET через debug-session клиента. Этот процесс отвечает за графический интерфейс и отображает частичный порядок выполнения программы. Чтобы построить это отображение, процесс debug-session ставит временные метки событиям, полученным от процесса disk server, использую данные, пересылаемые с каждым событием. Расстановка временных меток передана этому процессу для того, чтобы процесс disk server не записывал эту информацию в трассу.

Однако для вычисления временных меток логического времени (logical timestamps) требует просмотр истории выполнения программы с самого начала до того момента, когда рассматриваемое событие случится. Для того, чтобы вычисление времени произвольного события было более эффективно, есть третий процесс – checkpoint process. Этот процесс расставляет временные метки всем событиям и периодически сохраняет эту информацию. Затем, процесс debug-session находит последнюю контрольную точку, предшествовавшую событию, и запускает алгоритм вычисления времени события от последней контрольной точки. Таким образом, checkpoint process

существует только для улучшения производительности при вычислении временных меток событий.

Взаимодействие между debug-session и checkpoint процессами происходит через файл контрольных данных, а взаимодействие между процессами disk сервер и клиентами включает также посылку/прием сообщений.

К этой системе могут быть добавлены другие процессы. Например, таким образом, был добавлен процесс, определяющий предикаты. Пользователь может графически изобразить схему, по которой будут определяться предикаты.

Также с процессом disk server может связываться файл описания, который содержит таблицу описания символов. В этой таблице указывается по желанию пользователя то, как должны отображаться на экране те или иные события.

Как уже говорилось, POET позволяет повторно проигрывать трассы. При визуализации трассы выполнения программы на дисплее отображаются различные события. На любом из них позволяется поставить точку останова (breakpoint). При повторном выполнении программы процесс, на котором стоит точка останова, соответственно остановится нам этом месте. Затем с ним можно связать произвольный последовательный отладчик. Для других процессов пользователю позволяется выбрать способ остановки. Они могут быть остановлены либо в самом раннем консистентном состоянии, либо в самом позднем. Эти состояния определяются логическим временем – в самом раннем все события на других процессах должны предшествовать событию, на котором стоит точка останова. Соответственно, в самом последнем состоянии – наоборот.

POET был успешно адаптирован к различным системам. Среди них ABC++, uC++, Concert/C, OSF DCE, Hermes, PVM и SR.

**KOJAK**

(Kit for Oblective Judgement and Knowledge-based Detection of Performance Bottlenecks) [36] – набор инструментов, предназначенный для анализа выполнения параллельных программ. Его структура представлена на Рис. 13.

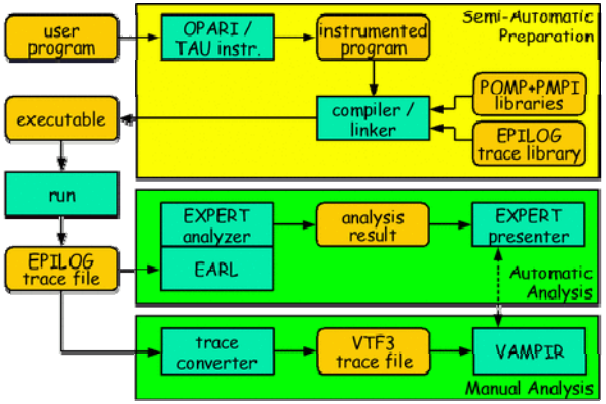


*Рис. 13. Архитектура KOJAK*



Сгенерированная трасса подается на вход Expert (Extensible Perfomance Tool). Expert – инструмент автоматического анализа трасс выполнения программ MPI и OpenMP. В свою очередь, Expert состоит из двух частей – анализирующей и визуализирующей. Анализирующая часть Expert использует EARL для доступа к трассе. В процессе анализа в трассе производится поиск шаблонов, заданных отдельной спецификацией. После анализа полученные результаты представляются с помощью визуализатора Expert. Типичный screenshot представлен на Рис. 14.

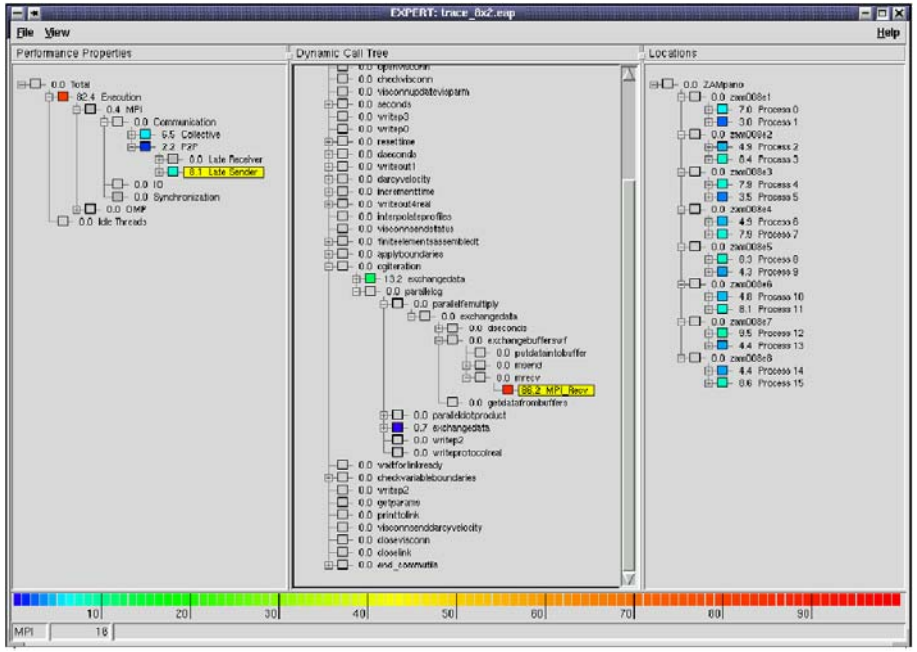


*Рис. 14. Главное окно Expert*

Слева показывается информация о том, какое именно событие найдено в программе, посередине – распределение этого события по дереву вызовов программы, справа – локализацию события (на какой машине, в каком процессе). Спецификация схем неэффективного выполнения программы представлена с помощью классов схем отдельно от Expert, что позволяет пользователю задавать свои собственные схемы, которые подвергаются анализу.

**Pablo**

Трассы формата SDDF [29] могут быть проанализированы с помощью Pablo Analysis GUI [30, 48]. Это набор модулей, взаимодействующих между собой. Analysis GUI предусматривает 4 основных типа модулей, выходные результаты которых могут быть показаны – анализ данных, представление данных, вход трассы и выход трассы. Также пользователь может самостоятельно определить

собственные модули, показывающие специфическую для программы информацию.

SvPablo (Source View Pablo) [48] – другой подход, в котором компоненты инструментирования, трассировки и анализа собраны вместе. После выполнения программы SvPablo GUI предлагает пользователю отображение статической информации для каждого инструментированного куска кода.

На рисунке приведен пример отображения информации в SvPablo (Рис. 15).

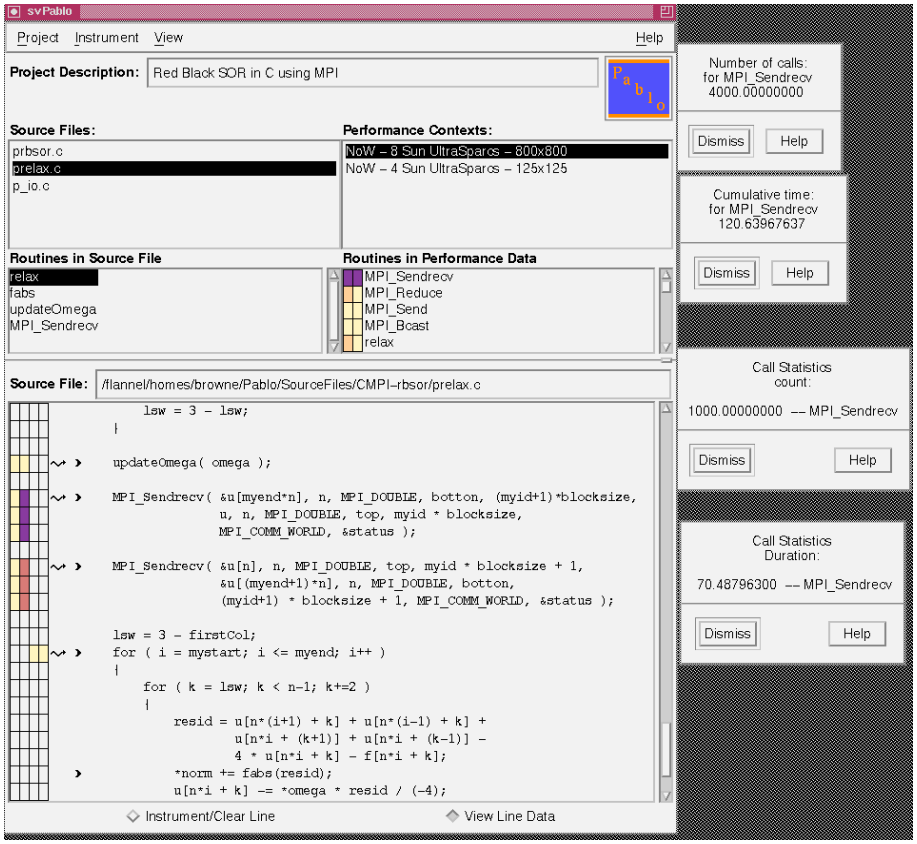


*Рис. 15. Главное окно SvPablo*

Также статистическая информация может быть показана с помощью модуля runSDDFStatistics. Кроме того, специальный модуль может конвертировать трассу в формате SDDF в XML для просмотра ее пользователем.

**VAMPIR и Intel Trace Analyzer**

Vampir [49, 48] – инструмент визуализации трасс, сгенерированных с помощью VampirTrace [34]. Vampir включает 3 основных категории визуального представления:



- `Process State Display` показывает каждый процесс прямоугольником и его состояние в один момент времени
- `Statistics Display` отображает суммарную статистику для всей трассы каждого процесса
- `Timeline Display` отображает изменение состояния процесса во времени и взаимодействие между процессами с помощью линий, их соединяющих

Отображения Process State и Statistics имеют как глобальную (для всех процессов), так и локальную форму (для одного процесса). В отображении Process State показывается состояние процесс – простой, соединение, выполнение. Отображение Timeline показывает, сколько времени процесс остается в одном и том же состоянии. Оно также имеет глобальную и локальную формы. Кроме этого, отображение может быть увеличено или уменьшено по желанию пользователя.

Выбрав графически какой-либо объект на дисплее можно узнать более подробную информацию о нем. Ниже приведен типичный screenshot Vampir (Рис. 16).

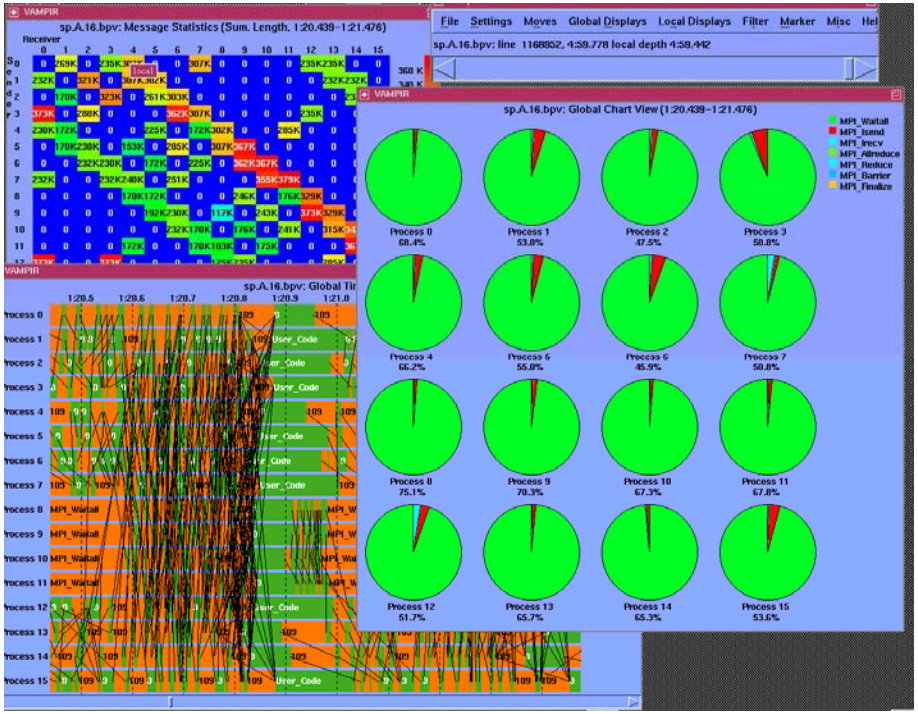


*Рис. 16. Главное окно Vampir*

Одним из недостатков Vampir является отсутствие связи отображений с исходным кодом. Однако можно вручную инструментировать программу, тем

самым, определив события или состояния, которые будут отображены на дисплее Timeline.

Кроме этого, Vampir допускает фильтрацию объектов, отображаемых на дисплее. Например, может отображаться только часть процессов, или сообщения только определенного типа могут быть показаны.

Intel Trace Analyzer обеспечивает близкую функциональность.

**XMPI**

В XMPI [35] можно записывать и воспроизводить трассы, а также изучать трассу выполнения программы. Во время выполнения программы ее можно остановить и проанализировать состояние. В отдельных окнах можно вывести информацию о состоянии процессов, сообщений, очередей сообщений. Активно используется цветовое отображение информации – цвет символа, соответствующего конкретному процессу, говорит о состоянии этого процесса – простой, выполнение, и т.д. Типичный screenshot xmpi показан на рисунке (Рис.17).

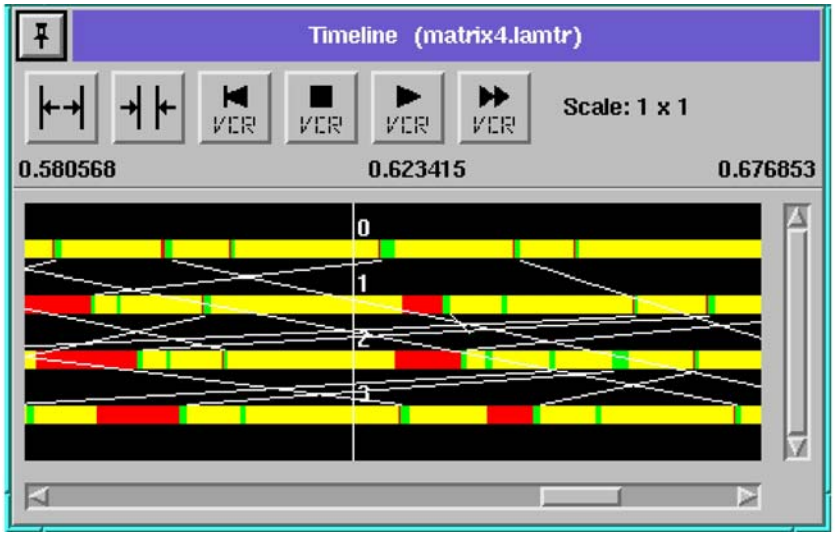


*Рис. 17. Временная диаграмма XMPI (физическое время)*

**Средства визуализации трасс MPE**

Сгенерированные с помощью MPE лог-файлы могут быть проанализированы различными средствами. Наиболее удобными средствами анализа являются графические средства визуализации: upshot, nupshot, jumpshot-2, jumpshot-3.

Самый простой вид upshot представлен на рисунке (Рис. 18). Каждая полоса соответствует процессу, различные участки полос окрашены в различные цвета соответственно тому, в каком состоянии находится процесс. Изображение может быть увеличено или уменьшено горизонтально или вертикально, центрировано относительно произвольной точки, и т.д. Инструмент nupshot использует устаревшую версию Tcl/Tk и поэтому не поддерживается. Jumpshot-2 и Jumpshot-3 разработаны на основе upshot и nupshot соответственно. Они написаны на Java и визуализируют двоичные файлы трасс. С их помощью можно визуализировать лог-файлы в различных представлениях. Пример такого представления, полученный с помощью Jumpshot-3, представлен на Рис. 19.



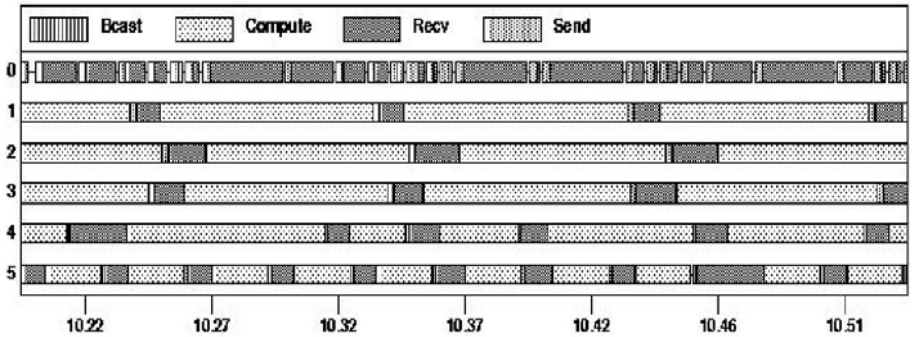


*Рис. 18. Временная диаграмма upshot*

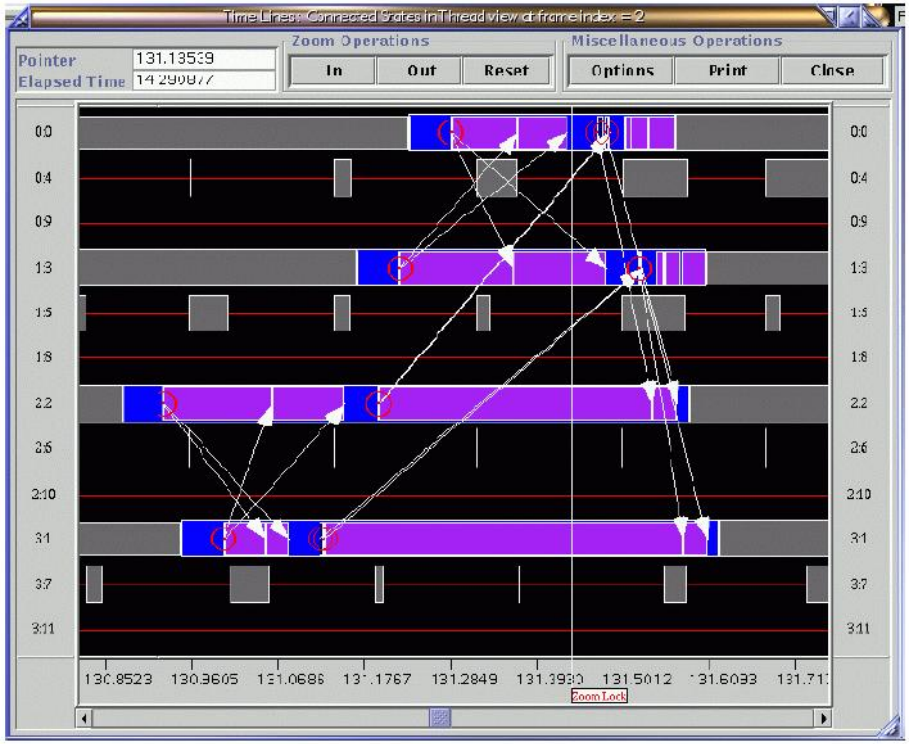


*Рис. 19. Временная диаграмма Jumpshot*

Кроме того, в MPE существует возможность связать наблюдаемую программу с анимационной библиотекой для анимации программы во время ее выполнения.

**MPVisualizer**

Инструмент для мониторинга и отладки параллельных программ MPVisualizer [9] содержит 3 компонента: trace/replay механизм, графическое представление и ядро визуализации. Одной из основных целей разработки MPVisualizer является возможность обнаруживать ошибки, связанные с недетерминизмом параллельной программы. Графический интерфейс отображает необходимую пользователю информацию: процессы, сообщения, события взаимодействия. Автоматически определяется в процессе анализа возможное состояние гонок.

Во время трассировки сохраняется только самая необходимая информация, чтобы минимизировать ее объем, но достаточная для того, чтобы гарантировать однозначность воспроизведения программы. Поэтому использование любого последовательного отладчика с любым процессом во время повторного выполнения не изменяет работу программы. Таким образом, допустима интеграция MPVisualizer с последовательными отладчиками.

Код мониторинга встраивается в стандартную библиотеку, которая не должна модифицироваться пользователем. Код приложения остается нетронутым, но связь с библиотекой его различна во время трассировки и повторного выполнения. Во время трассировки, приложение должно быть собрано с одной модифицированной библиотекой, trace library, во время повторного выполнения – с другой модифицированной библиотекой, replay library. Это позволяет гарантировать тот же самый порядок приема сообщений и то же самое выполнение программы во время повторного воспроизведения. Процессы посылают блоки данных специальному процессу–шпиону. К значимым событиям относятся не только события взаимодействия, но и начало, и конец процессов. Каждый раз при повторном воспроизведении, во время какого-либо события, блок данных (структура с полем, идентифицирующим тип события) посылается соответствующему процессу-шпиону, запущенному на той же самой машине. Затем этот процесс посылает блок данных так основному процессу, создавая своего рода мост между параллельной прикладной программой и объектно-ориентированной моделью программы.

MPVisualizer поддерживает наблюдение за программой во время повторного выполнения и во время посмертного анализа (post-mortem).

Механизм trace/replay был разработан и протестирован на основе библиотеки PVM, а двигатель визуализации и графический интерфейс – на C++ и Motif. Предполагается переносимость для MPI.

Кроме этого, в этом инструменте поддерживается определение состояния гонок. Как только это состояние определяется, пользователь информируется об этом соответствующим всплывающим окном с информацией о том, какие события и на каком процессе составляют это состояние гонок (Рис. 20).

Когда два события приема по шаблону случаются на одном процессе – это потенциально может означать возникновение состояния гонок. Однако есть



последовательности событий приема по шаблону, которые подпадают под это подозрение, но состоянием гонок не являются. Анализ частичного порядка позволяет устранить это. Если первый прием по шаблону случается до посылки сообщения, соответствующего второму приема по шаблону, в отношении порядка, установленного Lamport, то реального условия гонок не наблюдается.

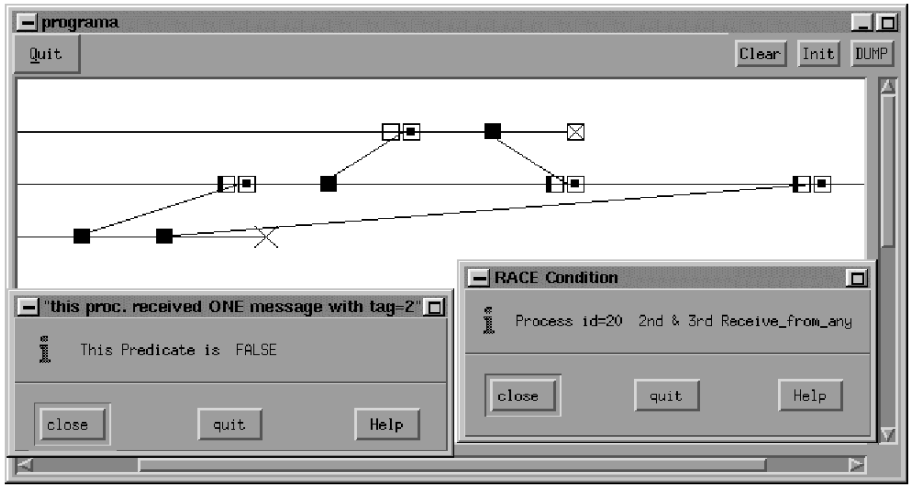


*Рис. 20. MPVisualizer – всплывающие окна, отображающие информацию о состоянии гонок и проверки выполнения предикатов.*

Также в MPVisualizer, кроме автоматического определения состояния гонок, есть автоматическая проверка предикатов – события связи, начало и конец процессов (Рис. 20). Вид предикатов, которые могут быть определены, зависит от степени детализации наблюдения. Если мы определяем блок как последовательность инструкций, выполняемых процессом между двумя значимыми событиями, предикат остается истинным или ложным в пределах этого блока и переоценивается всякий раз, когда его значение могло быть изменено. Пример предиката, который может быть обнаружен – "этот процесс принял точно одно сообщение с тэгом 2".

**Umpire**

Umpire – инструмент обнаружения ошибок времени выполнения MPI-программы [11]. С помощью профилировочного интерфейса PMPI, Umpire контролирует вызовы функции библиотеки MPI путем вставки прослойки между программой и библиотекой MPI. Его архитектура представлена на рисунке (Рис. 21).

Как видно из рисунка, Umpire состоит из двух частей – системы сбора и передачи информации и менеджера (сервера). Взаимодействие между ними происходит с использованием общей памяти, что ограничивает применимость инструмента. Проверки, выполняемые Umpire, делятся на две категории –

локальные и глобальные. Локальные проверки осуществляются локальной частью Umpire до передачи информации менеджеру (серверу). Примером локальной проверки является проверка контрольной суммы буфера неблокирующего сообщения. Затем данные о вызовах MPI-процедур передаются менеджеру, и он осуществляет глобальные проверки на наличие таких ошибок, как взаимоблокировки, некорректные коллективные операции, и т.д. Менеджер Umpire ведет историю событий, происходящих в системе. Информация о событии хранится до тех пор, пока не будет достоверно известно, что это событие не переводит систему в ошибочное состояние, а затем удаляется. Ошибки, отсутствие которых которые отслеживает Umpire, включают в себя взаимоблокировки, некорректные коллективные операции, ошибочную запись в посылаемый буфер, ошибки в работе с ресурсами (коммуникаторами, группами, и т.д.).

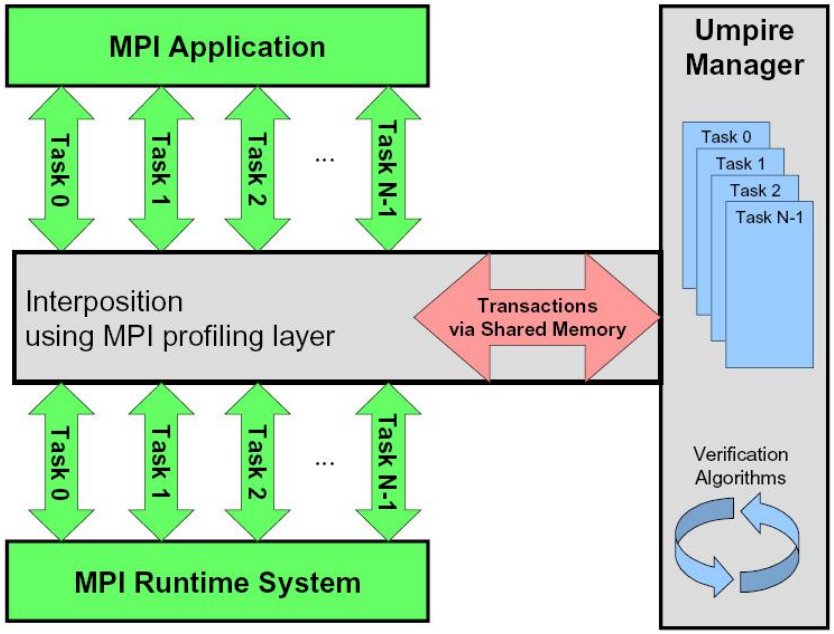


*Рис. 21. Архитектура Umpire*

Алгоритм определения взаимоблокировки, применяемый в Umpire, описан ранее (раздел 4). Следует заметить, что с помощью него можно проанализировать программу, использующую лишь часть MPI, так как он испытывает трудности, например, при работе с приемом по шаблону. Поэтому, кроме алгоритмического определения тупика, Umpire определяет в некоторых ситуациях тупики по тайм-ауту.

К некорректным операциям, которые определяет Umpire, относится также неправильный порядок вызова процедур-MPI (то есть, например, все процессы сначала вызывают MPI_Barrier, а потом MPI_Bcast, а один процесс наоборот,



сначала MPI_Bcast, а потом MPI_Barrier). Запись в буфер асинхронного сообщения после MPI_Isend, но до MPI_Wait также обнаруживается Umpire. К обнаруживаемым ошибкам можно отнести неправильную работу с ресурсами: их потерю, работу с неинициализированными ресурсами, и т.д.

**Marmot**

Инструмент Marmot [8] предназначен для обнаружения ошибок в MPI-программе во время её выполнения. Также как и Umpire, Marmot с помощью профилировочного интерфейса PMPI контролирует вызовы функции библиотеки MPI путем добавления прослойки между программой и библиотекой MPI. Его архитектура представлена на Рис. 22.

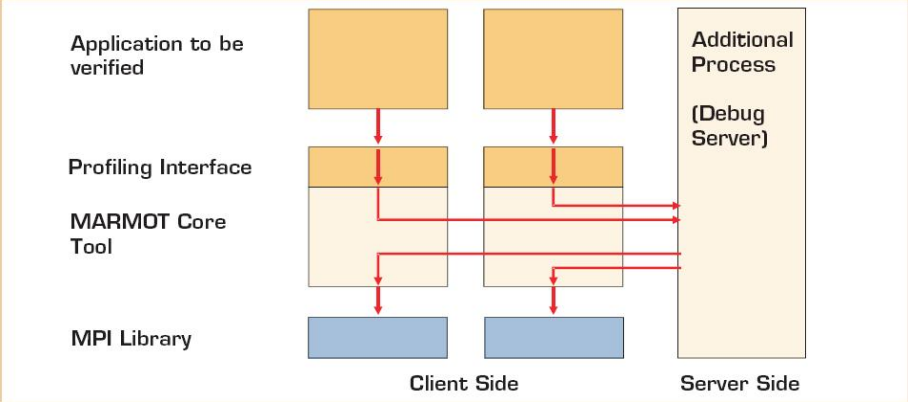


*Рис. 22. Архитектура Marmot*

Так же как и Umpire, Marmot состоит из двух частей – системы сбора и передачи информации и дополнительного процесса (сервера). Marmot добавляет дополнительный MPI-процесс для всех глобальных задач, которые не могут быть решены в пределах одного MPI-процесса, таких, как, например, обнаружение тупика. Другой глобальной задачей может быть сериализация выполнения программы, если эта опция выбрана пользователем. Локальные задачи, такие, как проверка работы с ресурсами, осуществляется на клиентской стороне. Информация между MPI-процессами и этим дополнительным серверным процессом передается с помощью MPI. Для программы пользователя этот процесс невидим. Достигается это путем отображения коммуникатора MPI_COMM_WORLD в коммуникатор Marmot, который содержит только MPI-процессы. Таким образом, в Marmot нет ограничения на применение в системах с общей памятью, как это есть в Umpire. Marmot отображает все коммуникаторы MPI в свои собственные, и, используя профилировочный интерфейс прерывает вызовы MPI-процедур с их параметрами для дальнейшего анализа. Затем Marmot проверяет корректность использования коммуникатора. Тоже самое происходит с группами, типами, операторами, и т.д.

Marmot определяет тупики следующим способом: дополнительный процесс считает, сколько времени каждый процесс ждет в вызове MPI процедуры. Если это время превышает пользователем заданный лимит, процесс помечается как повисший. Если все процесс повисли, то делается вывод о наличие тупика. Несколько последних MPI-вызовов до тупика могут быть трассированы назад для каждого процесса.

**MPI-CHECK**

MPI-CHECK [10, 50, 51] – инструмент для отладки MPI программ, написанных как на Фортране, так и на Си/Си++. MPI-CHECK автоматически проверяет программу на наличие следующих проблем:

- несоответствие типов аргументов;
- сообщения с выходом за границы массивов;
- сообщения с отрицательной длиной;
- вызовы MPI процедур до MPI_INITIALIZE или после MPI_FINALIZE;
- несоответствие между декларируемым типом сообщения и типом его аргумента;
- выявление взаимоблокировок.

На этапе инструментирования производится поиск и замена вызовов MPI в исходном тексте программы на вызовы функций MPI-CHECK, а также создается База Данных Программы (Program Data Base – PDB), содержащая информацию о каждом вызове MPI функций: ее имя, аргументы, номер строки и имя файла, в котором этот вызов был сделан.

Пример замены выглядит следующим образом:

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

заменяется на

```
X_MPI_Send(__FILE__,__LINE__, buf, count, datatype,
XDetectType( sizeof(buf), buf) , dest, tag, comm);
```

Поля __FILE__ и __LINE__ автоматически заполняются препроцессором. Функция XDetectType используется для определения Си++ типа для MPI буфера. Для определения типа посылаемого буфера MPI-CHECK использует следующий подход (для этого ему требуется C++ компилятор). Определяется набор функций с одинаковым именем, но различными типами входного параметра. В результате компилятор Си++ подставляет вызов соответствующей функции, которая возвращает тип буфера. Эта информация используется на этапе выполнения для сравнения с соответствующим типом MPI.

На этапе выполнения программы пользователь сначала запускает сервер MPI-CHECK, затем программу обычным способом. Найденные ошибки автоматически записываются сервером в выходной поток. Для выявления взаимоблокировок используется алгоритм поиска циклов в графе зависимостей процессов, а для некоторых случаев используется определение взаимоблокировки по тайм-ауту.



**Intel® Message Checker**

Intel® Message Checker (IMC) – инструмент отладки корректности MPI-программ, разработанный в Intel Advanced Computing Center [52]. IMC автоматически определяет некоторые типы ошибок в MPI-программе. Определяются такие ошибки, как несоответствие типов, состояние гонок, реальные и потенциальные взаимоблокировки, неправильная работа с ресурсами. IMC состоит из трёх компонентов:

- `Trace` Collector – трассировщик, собирает трассу;
- `Analyzer Engine` – анализатор, анализирует одну трассу или сравнивает две трассы;
- `Visualizer` – визуализатор, загружает трассу и предоставляет пользователю возможность обращаться к найденным ошибкам.

Первые две компоненты работают на кластере, Визуализатор работает на стороне клиента. Трассировщик использует стандартный интерфейс PMPI для сбора информации о вызовах MPI. Для каждого вызова он записывает все параметры вызова, а также дополнительную информацию (контрольная сумма, например), если это возможно. После завершения программы все трассы собираются вместе в один структурированный файл.

Анализатор ищет реальные или потенциальные ошибки в трассе. На выходе получается новый файл, содержащий информацию обо всех найденных ошибках в программе с соответствующими ссылками на трассируемые события. К ним относятся:

- несоответствие друг другу парных вызовов посылки и приёма сообщения;
- потенциальные или реальные взаимоблокировки;
- нестабильные ошибки, вызванные записью в посылаемый буфер до того, как посылка осуществилась;
- ошибочный порядок вызовов коллективных операций;
- нестабильные ошибки, вызванные специфическим перекрытием буферов в некоторых коммуникационных операциях, выполняемых параллельно на одном процессоре;
- несовпадение длин у посылаемого и принимаемого сообщений;
- несовпадение контрольных сумм у посылаемого и принимаемого сообщений;
- бесконечный цикл или аварийное завершение программы в MPI-вызове;
- несоответствие типов у посылаемого и принимаемого сообщений.

Визуализатор вначале показывает пользователю обобщенную информацию о типах найденных ошибок. Пользователь может подробнее рассмотреть описание ошибок интересующего его типа. Для каждой ошибки есть ссылки на источник (код), и есть возможность развернуть дерево вызовов. Кроме этого, есть другие дополнительные отображения.

**Средства отладки MPI-программ в DVM-системе**

Эти инструментальные средства [53] включают в себя:

- трассировщик;
- анализатор корректности;
- компаратор трасс;
- анализатор эффективности.

Трассировщик использует стандартный профилировочный интерфейс PMPI для сбора информации о вызовах функций MPI, об исключительных ситуациях при работе программы, и о заданных пользователем событиях.

Анализатор корректности обращений к MPI-функциям информирует пользователя о найденных ошибках. К ним относятся такие ошибки, как ненормальное завершение процессов, потенциальные и реальные тупики, отсутствие парных операций для не блокирующих операций обмена, некорректная работа с буферами сообщений, несовпадение при передаче сообщений контрольных сумм, длин, типов данных, некорректные коллективные операции, и т.д. Анализ корректности выполняется как во время работы программы, так и после её завершения по собранной трассе.

Компаратор трасс может осуществлять сравнительную отладку двумя способами – сравнивать выполняющуюся программу с трассой и сравнивать две ранее собранные трассы. Сравнительная отладка позволяет выявить такие несоответствия, как:отсутствие события в одной из трасс, изменение порядка событий, различие в переданных данных.

Анализатор эффективности предоставляет пользователю информацию о влиянии различных факторов, как на эффективность всей программы, так и различных задаваемых пользователем участков программы. Анализ эффек-тивности производится по собранной трассе, и, как и другая информация, полу-ченная в результате анализа, собирается в структурированный текстовый файл.

**Typed MPI**

Библиотека Typed MPI [54] предназначена для проверки типов в MPI. Она предоставляет пользователю интерфейс, заменяющий интерфейс MPI и передающий данные только тогда, если типы на передающей и принимающей сторонах совпадают. Пользователь должен использовать вместо функции MPI, например MPI_Send, функцию Typed MPI – MPI_Wrapper_Send. Пример замены приведен ниже:

```
MPI Call:    int MPI_Send (void* buf, int count,
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Typed MPI Call:    int MPI_Wrapper_Send (int dest,
int tag, MPI_Comm comm, (void*) buf1, int count1,…)
```

На принимающей стороне производится автоматическая проверка соответствия типов, и в обратном сообщении посылающая сторона извещается об успешности взаимодействия либо об ошибке.

Кроме проверки типов, Typed MPI предлагает пользователю интерфейс для автоматического конструирования из типов Си – типы MPI.



## 7.4. Инструменты для манипуляции поведением параллельной программы

**MAD**

Интегрированная среда для отладки параллельных программ MAD (Monitoring And Debugging environment) [38, 39] предоставляет пользователю возможность интерактивного выбора порядка рангов процессов, сообщения от которых принимаются по шаблону. Это осуществляется за счет выделения в трассе событий приема сообщений, порядок которых может быть изменен и обеспечения возможности их перестановки пользователем (Рис. 23). После перестановки трасса записывается, и программа воспроизводится вновь с измененной трассой до момента манипуляции, а далее выполняется без учета трассы.

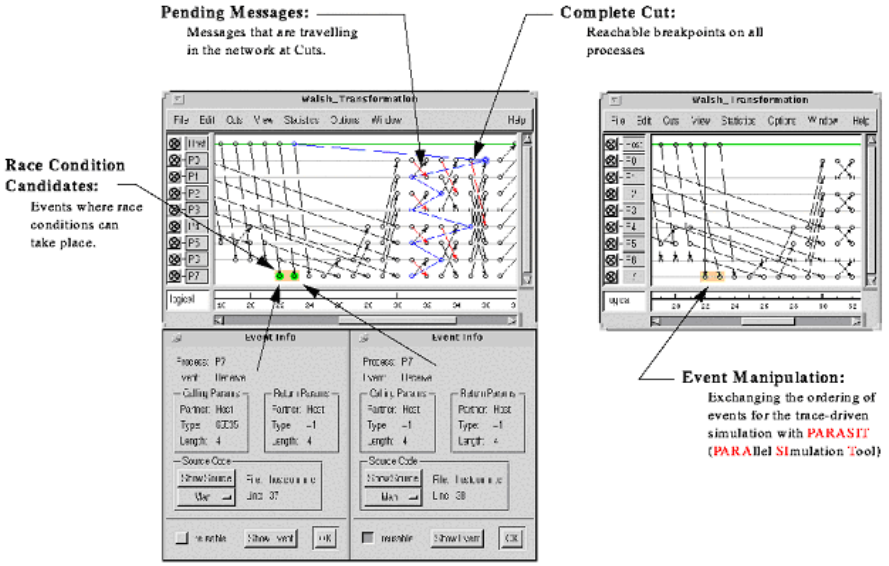


*Рис. 23. Возможность ручной манипуляции событиями в среде MAD*

**NOPE**

В работах D. Kranzlmüller и др. [40, 41] рассматривается система NOPE, предназначенная для автоматизации процесса отладки параллельных программ, основанных на посылке и приеме сообщений. Помимо функций мониторинга, в системе NOPE предусмотрена возможность автоматической манипуляции поведением программ с помощью перестановки событий приема по шаблону. Теоретические разработки подробно рассмотрены в разделе 6 данного обзора.

Отладка производится в два этапа. На первом этапе (record) инструментированная параллельная программа выполняется и производится сбор трассировочной информации. Фиксируется порядок, в котором сообщения принимались операцией приема по шаблону. На втором этапе (replay) программа выполняется и при этом фиксируется другой, отличный от

первоначального, порядок приема сообщения операцией приема по шаблону. Трасса опять собирается, возможно, при этом поменялся не только порядок но и появились новые события в ней. Процедура повторяется до тех пор, пока не будут перебраны все возможные варианты.

При трассировке записываются результаты вызовов функций (авторы предлагают реализацию такой трассировки только для функции rand), и последовательность рангов процессов, полученных функцией приема сообщений по шаблону. Этой информации достаточно для воспроизведения, в силу правила сохранения порядка сообщений.

На этапе воспроизведения выбираются возможные кандидаты для приема (racing messages) по шаблону. Это – все операции посылки, которые не завершились до момента, соответствующего функции приема. Выбирается некоторый порядок, в котором эти сообщения будут приняты. Программа вновь запускается с фиксацией порядка. При этом трасса может измениться, в ней могут появиться события, которых раньше не было, это потребует новых запусков.

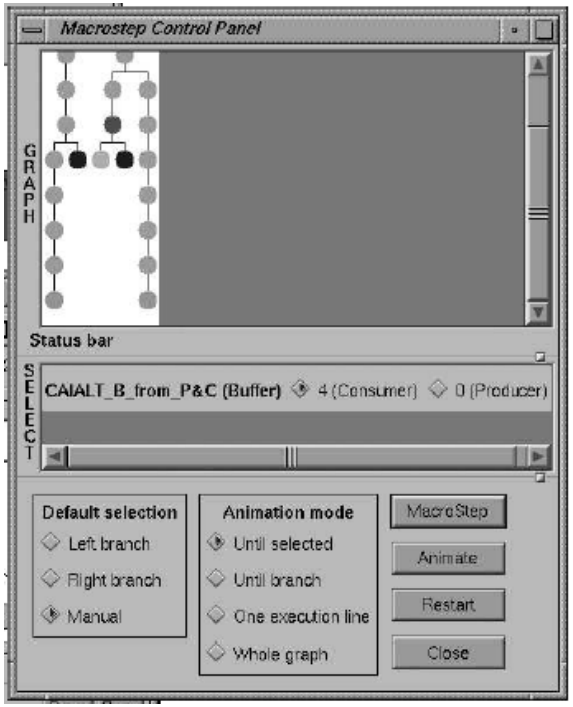


*Рис. 24. Графическое представление дерева обхода состояний параллельной программы в окне отладчика DIWIDE*



**DIWIDE**

Отладчик DIWIDE предоставляет пользователю возможности для пошагового выполнения программы и для систематического обхода дерева состояний. Подробное описание этих возможностей содержится в разделе 6. Дерево состояний программы визуализуется в специальном окне отладчика (Рис. 24), что позволяет пользователю интерактивно контролировать процесс обхода. Светлые круги соответствуют пройденным, а темные – состояниям, с которых начинаются не пройденные ветви дерева.

## 7.5. Интегрированные среды для отладки параллельных программ

**DeWiz**

В работе Schaubschlager [58] описана система DeWiz, предназначенная для анализа поведения параллельных программ. Эта система является развитием системы MAD и отличается от нее большей гибкостью и возможностями для расширения. Система DeWiz строится на основе соединения набора модулей, взаимодействие которых происходит с помощью протокола для передачи потока граф событий. Модульный подход делает систему гибкой и легко расширяемой. Пользователю предоставляется возможность выбирать и объединять выбранные модули во взаимодействующую систему, а также разрабатывать и подключать свои собственные модули. Набор модулей определяется структурой отлаживаемого приложения и задачами отладки, которые ставит пользователь. Модульная структура построенной пользователем системы представляется в графическом виде как набор связанных компонент (Рис. 25).

В системе DeWiz предусмотрены три типа модулей: модули генерации графа событий, анализа и визуализации. Модули генерации графа событий программы предназначены для построения графа событий по трассировочной информации, получаемой в результате выполнения программы. При этом возможны два режима работы: сбор информации во время работы программы и использование трасс, собранных в результате выполнения.

Модули, предназначенные для автоматизированного анализа ошибок в программе, работают с построенным графом событий. Производится анализ ошибок взаимодействия, проявляющихся как несовпадение по длине посланного сообщения и принятого сообщения. Также производится выявление шаблонов взаимодействия, характерных для ошибочного поведения и повторяющихся участков трасс, которые обычно соответствуют циклическим участкам программ. Последнее позволяет сжимать и эффективно визуализировать трассу.

Наконец модули визуализации предназначены для того, чтобы в удобном графическом виде представить граф событий и указать возможные аномалии. Модули взаимодействуют между собой по определенному протоколу, что

позволяет распределять различные модули на разные вычислительные узлы и производить анализ параллельно с работой приложения и мониторингом.

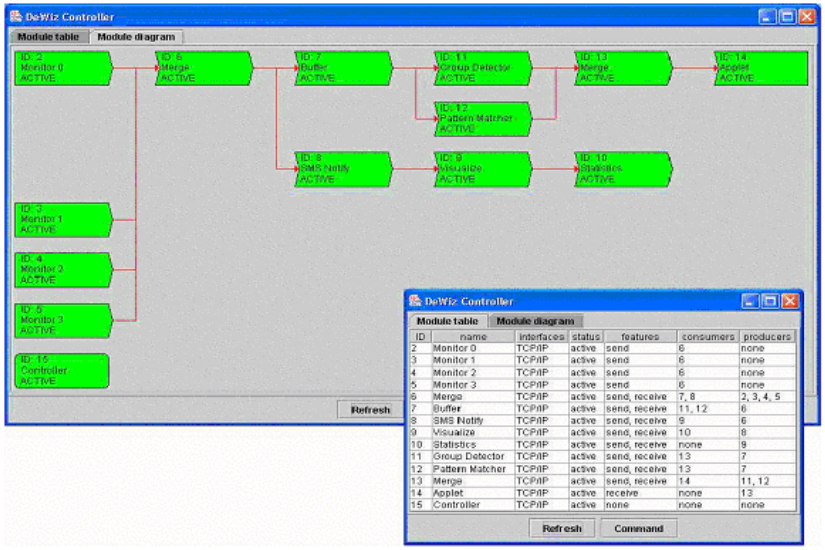


*Рис. 25. Пример графического представления взаимодействующих модулей системы DeWiz*

## 8. Заключение

На основании рассмотренного в обзоре материала можно сделать ряд выводов о степени решенности проблем, сформулированных во введении к обзору, которые приводятся далее.

Для решения задачи сбора трассы и повторного воспроизведения хода выполнения программы по этой трассе предложены ряд подходов, применимых к отладке реальных параллельных программ. Трассировка реализована во многих инструментах для автоматического анализа параллельных программ, а для хранения трассы разработаны ряд стандартных представлений. Можно сказать, что для данной проблемы имеется достаточно полный набор решений, прошедших апробацию на практике.

Задаче визуализации, как правило, рассматривается в практической плоскости. Теоретические исследования в этой области сталкиваются со значительными трудностями, связанными с формализацией постановки задачи. К настоящему моменту предложено большое число различных способов визуализации, предоставляющих пользователю разнообразные возможности для наблюдения и интерактивного контроля над ходом выполнения программы.

Исследованию задачи анализа посвящено большое число работ и соответствующие модули реализованы в инструментальных средствах.



Несмотря на достаточно широкий спектр предлагаемых подходов, существует ряд открытых проблем. В частности, отсутствует универсальное решение для проблемы обнаружения ситуации взаимоблокировки. Методы анализа, предлагаемые для MPI-программ, рассматривают подмножество функций библиотеки, многие сложные типы взаимодействий не исследуются. Рассмотрение ограничено только стандартом MPI-1.

Недостаточно исследован вопрос об интеграции методов анализа, разработанных для последовательных программ и методов для параллельных программ. Вместе с тем для выявления таких нарушений семантики выполнения параллельной программы как выход значения выражения за границы типа данных и ошибок, связанных с некорректной работой с динамической памятью, целесообразно совмещать использование этих методов.

В наименьшей степени достигнуты продвижения в области автоматического перебора вариантов выполнения параллельной программы. Имеющиеся теоретические разработки предполагают полный перебор возможных реализаций приема по шаблону. Это делает затруднительным использование предлагаемых подходов для отладки реальных промышленных приложений, где количество возможных вариантов таково, что затраты на их перебор заведомо превышают любые потенциально-доступные вычислительные ресурсы. Не разработаны системные критерии и методы для сокращения такого перебора.

Подводя итог обзору можно сказать, что методы автоматизации мониторинга, анализа и манипуляции поведением параллельных программ в настоящее время активно развиваются. Несмотря на значительный прогресс, достигнутый в этой области, остается большое количество открытых проблем, решение которых является актуальной и важной как в теоретическом, так и в практическом отношении задачей.

Авторы выражают благодарность старшему научному сотруднику ИСП РАН Алексею Яковлевичу Калинову за постановку задачи и плодотворные обсуждения материала обзора, а также ведущим научным сотрудникам ИСП РАН Александру Сергеевичу Косачеву и Игорю Борисовичу Бурдонову за внимание к работе и полезные дискуссии. Авторы глубоко признательны зав. отделом программных комплексов ИПМ им. М.В. Келдыша РАН Виктору Алексеевичу Крюкову за предоставленные материалы, вошедшие в настоящий обзор, и полезные замечания к его содержанию.

**Литература**

1. M.S. Otto, S. Huss-Liderman, D. Walker, J.Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.
2. A. Geist, A. Beguelin, J.Dongarra, W. Wiang, R. Manchek, V.Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
3. Г. Эндрюс. *Основы многопоточного, параллельного и распределенного программирования*. Издательский дом "Вильямс", 2003.
4. В.В. Топорков. *Модели распределенных вычислений.* ФИЗМАТЛИТ, 2004.

5. L. Lamport. *Time, clocks and the ordering of events in a distributed system.* Communications of ACM, 21(7):558-565, 1978.
6. C. Fidge. *Logical time in distributed computing systems*. IEEE Computer, 24(8):28-33, 1991.
7. D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Joh. Kepler University Linz, September 2000, http://www.gup.uni-linz.ac.at/~dk/thesis.
8. B. Krammer, K. Bidmon, M. Müller, M. Resch. *MARMOT: An MPI analysis and testing tool*. Proc. of Parallel Computing 2003 (PARCO2003), 493-500, Elsevier, 2003.
9. P. Claudio, J. Cunha, M. Carmo. *Monitoring and Debugging Message Passing Applications with MPVisualizer*. Proc. of 8th Euromicro Workshop, 376-382, 19-21 January, 2000.
10. G. Luecke, Y. Zou, J. Coyle, J. Hoekstra, M. Kraeva. *Deadlock detection in MPI programs.* Concurrency and Computation: Practice and Experience, 14(11):911-932, 2002.
11. J. Vetter, B. Supinski. *Dynamic software testing of MPI applications with Umpire.* Proc. of SC2000, ACM/IEEE, 2000.
12. K. Chandi, L. Lamport. *Distributed snapshots: Determining global states of distributed systems.* ACM Transactions on Computer Systems, 3(1):63-75,1985.
13. N. Thoai, D. Kranzlmüller, J. Volkert. *Shortcut replay: A replay technique for debugging long-running parallel programs.* Proc. of ASIAN 2002, volume 2250 of LNCS, 664-67, Springer-Verlag, 2002.
14. N. Thoai, D. Kranzlmüller, J. Volkert. *Rollback-one-step checkpointing and reduced message logging for debugging message-passing programs*. Proc. of International Meeting on Vector and Parallel Procesing (VECPAR'2002), 2002.
15. P. Kacsuk. *Systematic debugging of parallel programs in DIWIDE based on collective breakpoints and macrosteps.* Proc. of International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1999), 83-96, 1999.
16. P. Kacsuk, R. Lovas, J. Kovacs. Systematic debugging of parallel programs in DIWIDE based on collective breakpoints and macrosteps. Proc. of Euro-Par'99, volume 1685 of LNCS, 90-97, Springer-Verlag, 1999.
17. Д. Пелед, Э.М. Кларк, О. Грамберг. *Верификация моделей программ: Model Checking*. МЦНМО, 2002.
18. J. Kovacs, G. Kusper, R. Lovas, W. Schreiner. *Integrating temporal assertions into a parallel debugger.* Proc. of Euro-Par 2002, 113-120. Springer-Verlag, 2002.
19. Y. Chan, W. Gropp, E. Lusk. *User's Guide for MPE: Extensions for MPI Programs*. www.mcs.anl.gov/mpi/mpich/docs/mpeguide/paper.htm.
20. Г.И. Шпаковский, Н.В. Серикова, *Программирование для многопроцессорных систем в стандарте MPI.* БГУ, Минск, 2002.
21. G.A. Geist, M.T. Heath, B.W. Peyton, P.H.Worley, *A Users' Guide to PICL- A Portable Instrumented Communication Library*. ORNL/TM- 1616, Oak Ridge National Lab, Tennesee, USA, August 1990, http://www.epm.ornl.gov/picl/.
22. P.H. Worley. *A New PICL Trace File Format*. Tech. Rep. ORNL/TM-12125, Oak Ridge National Laboratory, Tennesee, USA, September 1992, http://www.epm.ornl.gov/picl/.
23. F. Wolf, B. Mohr. *EPILOG Binary Trace-Data Format*. Tech. Report FZJ-ZAM-IB-2004-06, Forschungszentrum, Julich, May, 2004.
24. F. Wolf, B. Mohr. *EARL – A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*. Proc. of the 7th International Conference on High-Performance Computing and Networking (HPCN'99), 503-512, Amsterdam(The Netherlands), April 12-14, 1999.



25. F. Wolf, B. Mohr. Automatic Performance Analysis of MPI Applications Based on Event Traces. Proc. of the 6th international Euro-Par Conference on Parallel Processing, 123-132,vol. 1900 of LNCS, Springer-Verlag, London, 2000.
26. F. Wolf. *EARL – Event Analyzis and Recognition Language: Reference Manual*. Tech. Report ZAM-IB-0002, Julich, Germany, February 2000.
27. F. Wolf, B. Mohr. *Automatic performance analysis of hybrid MPI/OpenMP applications.* Journal of Systems Architecture, 49(10-11):421–439, November 2003.
28. F. Wolf. *EARL – API Documentation*. Tech. Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory, October 2004.
29. R. Aydt. *The Pablo Self-Defining Data Format*. Tech. Report, University of Illinois at Urbana-Champaign, March 1992.
30. D.Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, L. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. Proc. of the Scalable Parallel Libraries Conference, IEEE Computer Society, 1993.
31. *Intel Trace Collector. User's Guide.* Intel GmbH, Bruhl, Germany, January 2004, http://www.intel.com/software/products/cluster/tcollector/sysreq.htm.
32. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. *VAMPIR: Visualization and analysis of MPI resources.* Supercomputer, 12(1): 69-80, January 1996.
33. *Vampir. User's Guide.* PALLAS GmbH, Bruhl, Germany, January 2004, http://www.pallas.com.
34. *Vampirtrace. User's Guide.* PALLAS GmbH, Bruhl, Germany, January 2004, http://www.pallas.com.
35. N. J. Nevin. *The XMPI API and trace file format.* January 29, 1997, http://www.mpi.nd.edu/lam/
36. F. Wolf, B. Mohr. *KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Programs.* Proc. of the International Conference on Parallel and Distributed Computing (Euro-Par 2003), Klagenfurt, Austria, August 2003.
37. M. T. Heath, J. E. Finger. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs.* Oak Ridge National Laboratories, 1993, http://www.netlib.org/picl/.
38. Kranzlmüller, Ch. Schaubschlaeger, J. Volkert. *A Brief Overview of the MAD Debugging Activities.* Proc. of Fourth International Workshop on Automated Debugging (AADEBUG 2000), Munich, 2000.
39. Kranzlmüller, A. Rimnac. *Parallel Program Debugging with MAD – A Practical Approach.* Proc. of International Conference on Computational Science, 201-212, Melbourne, Australia, June 4, 2003.
40. D. Kranzlmüller, J. Volkert. Debugging point-to-point communication in MPI and PVM. Proc. of PVM/MPI'98. volume 1497 of LNCS, 265-272, Springer-Verlag, 1998.
41. D. Kranzlmüller, J Volkert. NOPE: A nondeterministic program evaluator. vol. 1557 of LNCS, 490-499, Springer-Verlag, 1999.
42. T. Kunz and D. J. Taylor. *Visualizing PVM executions.* University of Waterloo, Proc. of the 3rd PVM User's GroupMeeting,May,Pittsburgh,PA.,1995, http://www.cs.cmu.edu/Web/Groups/pvmug95.html
43. M. Khouzam , T. Kunz. *Single stepping in event-visualization tools.* Proc. of the 1996 conference of the Centre for Advanced Studies on Collaborative research, November 12-14, 1996.
44. T. Kunz, D. J. Taylor, J. P. Black. *Poet: Target-System-Independent Visualizations of Complex Distributed Executions.* HICSS (1), 452-461, 1997, http://csdl.computer.org/comp/proceedings/hicss/1997/7734/01/7734010452abs.htm.
45. T. Kunz. *Abstract behaviour of distributed rxecutions with rpplications to visualization.* PhD thesis, Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany, May 1994.

46. Y.-K. Yu, *Integrating Event Visualization with Sequential Debugging.* essay requirement for the degree of Master of Mathematics in Computer Science, Waterloo, Ontario, Canada, 1996.
47. M. Khouzam, *Single Stepping in Events Visualization Tools for Distributed Applications.* Thesis requirement for the degree of Master of Mathematics in Computer Science, Waterloo, Ontario, Canada, 1996.
48. S. Moore, D. Cronk, K. S. London, J. Dongarra. *Review of Performance Analysis Tools for MPI Parallel Programs.* Proc. of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, September 23-26, 2001, http://www.cs.utk.edu/browne/perftools-review/
49. H. Brunst, D. Kranzlmüller, W.E. Nagel. *Tools for Scalable Parallel Program Analysis – Vampir VNG and DeWiz.* Proc. of Distributed and Parallel Systems: Cluster and Grid Computing (DAPSYS 2004), 93-102,September 19-22, 2004.
50. Y. Chen, J. Coyle, J. Hoekstra, M. Kraeva, Y. Zou. *MPI-CHECK: a Tool for Checking Fortran 90 MPI Programs.* Concurrency and Computation: Practice and Experience, 15:93-100, 2003.
51. R. Luecke, P. Krusina. *MPI-CHECK for C/C++ MPI programs.* Iowa State University, preprint, November 13, 2003, http://www.public.iastate.edu/~grl/santafe.ps
52. J. DeSouza, B. Kuhn, B. R. Supinski. *Automated, scalable debugging of MPI programs with Intel® Message Checker.* SE-HPCS'05, May 15, 2005.
53. К.Н. Ефимкин, В.А. Крюков, В.Н. Ильяков, Ю.Л. Сазанов, В.Ф. Алексахин, М.И. Кулешова. *Средства отладки MPI-программ в DVM-системе.* Труды конференции Научный сервис в сети Интернет: технологии распределенных вычислений, Абрау-Дюрсо, 19-24 сентября 2005.
54. N. Bahadur, F.I. Popovici. *Typed MPI – A Data Type Tool for MPI.* 1999, http://www.cs.wisc.edu/~bnitin/736/project.doc.
55. Ch. Schaubschlager, D. Kranzlmüller, J. Volkert. *Event-based program analysis with DeWiz.* Proc. of CoRR'2003, http://arxiv.org/abs/cs.SE/0310007.