УДК 004.052.3

ОБЕСПЕЧЕНИЕ ОТКАЗОУСТОЙЧИВОСТИ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЙ С ПОМОЩЬЮ ЛОКАЛЬНЫХ КОНТРОЛЬНЫХ ТОЧЕК[1]

А.А. Бондаренко, М.В. Якобовский

Рассматриваются вопросы, связанные с проведением расчетов в распределенных вычислительных системах, компоненты которых подвержены отказам. В работе приводятся: определения системы, сбоя, ошибки, отказа и модели сбоя; наиболее важные результаты исследований отказов в параллельных вычислительных системах, в том числе с большими группами дисков; основные существующие методы восстановления и распространенные программные реализации обеспечения отказоустойчивости. Развивается подход обеспечения отказоустойчивости на уровне пользователя. Данный подход требует непосредственного участия разработчика прикладной программы в реализации метода обеспечения отказоустойчивости, в частности в формировании контрольных точек и процедур восстановления. Предложена схема сохранения в памяти вычислительных узлов данных прикладной программы, формирующих согласованную глобальную контрольную точку. В её рамках осуществляется дублирование локальных контрольных точек, что позволяет восстановить вычислительный процесс, если число отказов не превосходит допустимого для данной схемы уровня. Она может быть использована в различных протоколах восстановления и их модификациях.

Ключевые слова: параллельные вычисления, отказоустойчивость, контрольные точки, MPI.

Введение

Современные суперкомпьютеры состоят из десятков тысяч узлов, каждый из которых оснащен процессорами и, как правило, различными ускорителями. Увеличение количества компонентов системы, без существенного изменения элементной базы или технологии производства, естественным образом ведет к увеличению вероятности отказа некоторых компонентов системы [1, 2], т.е. приводит к тому, что время наработки на отказ вычислительной системы становится мало и сравнимо со временем проведения сеанса расчета. Оценки исследователей в суперкомпьютерной области показывают, что время наработки на отказ может составлять всего несколько часов [3].

В области аппаратного обеспечения активно разрабатываются пути уменьшения вероятности отказа компонент вычислительных систем [4]. Однако невозможно создать стабильное устройство, в особенности такое сложное как память, процессор или коммуникационное оборудование, поэтому важным является разработка программных средств обеспечения отказоустойчивости. С этой точки зрения существенное значение имеют методы восстановления, основанные на контрольных точках. Подобные методы предполагают периодическое сохранение состояния системы в централизованное устройство хранения, что позволяет при выходе из строя части вычислительного поля продолжить расчет с последней сохранённой согласованной глобальной контрольной точки [5-7].

[1] Статья рекомендована к публикации программным комитетом XIII Всероссийской научной конференции «Высокопроизводительные параллельные вычисления на кластерных системах» 2013г.

Главной проблемой масштабируемости технологии создания контрольных точек для современных вычислительных систем является ограниченная пропускная способность централизованного устройства хранения. Для широко распространенного координированного протокола характерной является одновременная запись локальных контрольных точек, приводящая к повышенной загруженности узлов ввода/вывода и централизованного устройства хранения. Некоторыми исследователями [2] отмечается, что время сохранения согласованной глобальной контрольной точки для систем Терафлопсного и Петафлопсного уровня составляет порядка 20-30 минут. Фактически, в перспективе, с ростом числа компонент системы и с уменьшением времени наработки на отказ всей системы, сама возможность успешного сохранения согласованной глобальной контрольной точки окажется под вопросом [2, 8].

При проведении высокопроизводительных вычислений представляется важным решение следующей задачи: разработать принципы сохранения контрольных точек за время, меньшее характерной продолжительности безотказной работы системы, и алгоритмы, обеспечивающие, в случае отказа части оборудования, быстрое автоматическое возобновление расчета на работоспособной части вычислительного поля. В данной работе, на уровне пользователя [1], развивается подход обеспечения отказоустойчивости высокопроизводительных вычислений, требующих регулярных обменов данными между процессами. Этот подход позволяет сократить объем контрольных точек ценою усложнения программного кода, поскольку предполагает непосредственное участие разработчика прикладной программы в реализации метода обеспечения отказоустойчивости (в формировании контрольных точек и процедур автоматического восстановления). В работе представлена схема сохранения локальных контрольных точек в память вычислительных узлов (оперативная, дисковая и т.п.), к которым имеют доступ один или группа MPI-процессов. Данная схема сохранения позволяет произвести восстановление процесса вычислений при возникновении нескольких отказов в MPI-процессах, вызванных отказами в вычислительных узлах.

Первый раздел статьи посвящен описанию отказов в вычислительных системах. В разделе 2 приведены сведения о методах и средствах обеспечения отказоустойчивости в распределенных системах. В последнем разделе изложена схема сохранения локальных контрольных точек.

1. Отказы в распределенных системах

Прежде чем говорить об отказоустойчивости, приведем основные определения и рассмотрим характеристики отказов в распределенных системах.

1.1. Сбои, ошибки и отказы

Приведем описание понятий «система», «сбой», «ошибка» и «отказ» согласно работам [9, 10], принятыми в области гарантированных вычислений представителями научной и технической среды.

Система является объектом, который взаимодействует с другими объектами, то есть другими системами. В роли систем могут выступать, в том числе, аппаратное обеспечение, программное обеспечение, люди и физический мир с его природными явлениями.

Для каждой системы есть другие системы, которые являются внешними по отношению к данной системе, то есть образуют окружающую среду. Система состоит из набора компонент, каждая из которых, в свою очередь, сама является системой, со своей внутренней структурой. Системы, входящие во множество взаимодействующих систем уровня n, являются подсистемами для системы уровня $n+1$. Каждая система уровня n состоит из множества подсистем уровня $n-1$, которые в свою очередь состоят из подсистем уровня $n-2$ и т.д. Эта иерархия систем и подсистем продолжается до уровня, для которого либо не возможна, либо не имеет смысл дальнейшая детализация на подсистемы. Подсистемы на этом последнем уровне называются атомарными компонентами, а сам уровень определяется в зависимости от рассматриваемой задачи.

В работе рассматриваются компьютерные и коммуникационные системы. Данные системы являются искусственными, их функции определяются соответствующими спецификациями. Поведением системы является то, что система делает, чтобы реализовать свою функцию. Корректным называется поведение, обеспечивающее реализацию функции системы.

Со временем состояние некоторых компонент системы может отклониться от состояний, соответствующих корректному поведению системы. Это измененное состояние компонент системы называется ошибкой (error). Наличие ошибки не означает, что поведение системы не может реализовать свою функцию. Однако, может наступить такое событие, когда осуществляемое поведение отклоняется от корректного, то есть система не реализует ожидаемую функцию. В этом случае говорят, что наступил отказ системы (system failure). Пусть возникла ошибка, тогда существует последовательность действий, выполнение которых приведет к отказу системы, если не будут предприняты корректирующие меры. Установленная или предполагаемая, причина ошибки называется сбоем (fault). Таким образом, сбой вызывает ошибку, которая может и не привести к отказу системы.

1.2. Модели сбоев

Такая абстракция как «модель сбоя», имеет существенное значение для разработки методов обеспечения отказоустойчивости. Приведем описание модели сбоев согласно работе [10].

Пользователь рассматривает конкретный уровень системы – уровень вентилей и логических схем, уровень функциональных блоков, уровень микросхем и т.п. В большинстве случаев пользователя не интересуют отказы на нижних уровнях системы. Для него важен факт наличия отказа на том уровне, на котором он рассматривает данную систему, как следствие сбоя на одном из нижних уровней. Один из способов удовлетворить интерес пользователя заключается в описании, для более высокого уровня, эффекта сбоя, произошедшего на нижнем уровне. Абстракция, заключающаяся в определение эффекта сбоя для конкретного уровня, называется моделью сбоя (fault model). Модель сбоя фактически описывает нарушения в функционировании системы, то есть отказы для конкретного уровня, к которым могут привести различные сбои на нижних уровнях. Обычно многие сбои на нижних уровнях могут приводить к одинаковым нарушениям функционирования системы, то есть быть представлены одной моделью сбоя на более высоком уровне. Пользователь выбирает модель сбоев так, чтобы она описывала наиболее важные сбои на нижних уровнях. Если такая модель существует, то пользова-

телю, на интересующем его уровне системы, необходимо учитывать отказы только в выбранной модели сбоев.

Модели сбоев, используемые в настоящее время, значительно варьируются: от очень подробных (на уровне транзисторов), до функционального уровня устройств. Наиболее часто в программных средствах обеспечения отказоустойчивости в качестве модели сбоя на функциональном уровне принимается либо модель «поломка» (crash failure), либо модель «аварийная остановка» (fail-stop failure) устройства вычислительной системы. Здесь под «устройством вычислительной системы» понимается процессор, ускоритель, коммутатор, модуль оперативной памяти, жесткий диск и т.п. Отличие между этими моделями заключается в том, что для модели «поломка» информация о неработоспособности устройства не передается другим компонентам системы, а для модели «аварийная остановка» информация распространяется среди соответствующих компонентов системы.

1.3. Статистика отказов

Создание новых суперкомпьютеров сопряжено с необходимостью решения ряда задач [8], среди которых важное место занимает возможности выполнения расчетов, несмотря на отказ ряда задействованных в расчете вычислительных узлов. Для выработки лучшей стратегии обеспечения отказоустойчивости необходимо иметь набор экспериментальных данных позволяющих делать выводы о статистике отказов в определенных вычислительных системах. Одним из шагов к появлению данных исследований является создание базы отказов. Так LANL (Los Alamos National Laboratory) и NERSC (National Energy Research Scientific Computing Center) опубликовали накопленные данные о сбоях в используемых ими системах [12, 13].

В работе [14] проведен анализ отказов, описанных в 10 наборах данных сроком сбора от 5 до 18 месяцев каждый, полученных с 8 вычислительных систем, на которых решались различные задачи. Результаты работы [14] расходятся с результатами, представленными в других источниках, что дает еще одно подтверждение необходимости создания обширной общедоступной базы отказов. Существование подобной базы отказов даст возможность для каждого исследователя сравнить свои результаты и методики оценки характеристик отказов с результатами других исследований, проведенных на одних и тех же наборах данных.

Существуют различные показатели характеризующие надёжность вычислительных систем [6, 7]. Одними из существенных для анализа отказов являются MTBF и MTBI – среднее время между отказами и между прерываниями. Значения этих показателей для некоторых распределенных вычислительных систем представлены в работе [3]. Среднее время между отказами/прерываниями составляет от 6.5 до 40 часов, а наиболее частыми отказами были: отказы в процессоре и устройствах хранения.

При анализе отказов важным является определение процентного соотношения между категориями отказов. В работе [15] выделили следующие категории отказов: аппаратные, программные, сетевые, связанные с человеческим фактором, связанные с внешним обеспечением работы вычислительной системы (перебои электричества). В результате исследований отказов зафиксированных в 1996-2005 гг. на группе вычислительных систем LANL было выявлено, что аппаратные отказы составляют более 60% от всех зарегистрированных отказов.

В работе [16] проведено исследование, охватывающее в общей сложности более 100 000 жестких дисков, как минимум от четырех различных поставщиков. Наборы данных об отказах собирались от 1 месяца до 5 лет каждый и получены на нескольких крупных высокопроизводительных вычислительных системах. В [16] получены следующие результаты. Ежегодно заменяется более 1% дисков (обычно от 2 до 4% вплоть до 13%). Были найдены свидетельства, что интенсивность отказов в дисках не является постоянной от времени эксплуатации. Наблюдалось раннее начало деградации дисков; доля замененных дисков, постоянно росла с возрастом, хотя часто предполагается, что этот эффект не проявляется до окончания номинального срока службы 5 лет. В итоге авторы работы [16] предположили, что правила замены дисков должны отличаться от текущего описания в спецификациях производителей дисков.

Из приведенных результатов исследований в области отказов больших групп дисков следует, что для распределенных систем Петафлопсного уровня и выше вероятность отказа узла значительна из-за высокой вероятности отказа его компонент, таких как диски, процессоры, память и другие.

2. Обеспечение отказоустойчивости для распределенных систем

Далее рассматриваются теоретические методы восстановления и распространенные программные решения в области обеспечения отказоустойчивости при проведении высокопроизводительных вычислений, характеризующихся выполнением больших объемов вычислений на большом количестве узлов в течение длительного времени (порядка десятков и более часов).

2.1. Методы восстановления

Под отказоустойчивостью понимают способность системы выполнять работу даже при наличии отказов [9]. Рассмотрим методы обеспечения отказоустойчивости в распределенных вычислительных системах, обеспечивающие продолжение расчета после отказа части вычислительных узлов.

Методы восстановления (rollback-recovery) [5 – 7] работы системы после возникновения отказов принято разделять на методы прямого (forward recovery) и обратного (backward recovery) восстановления. Первые восстанавливают систему в безошибочное состояние на основе текущих данных и по результатам анализа отказа, без обращения к предыдущим состояниям системы. Методы прямого восстановления основаны на специальных алгоритмах, например, широко известны стабилизирующиеся алгоритмы [11]. Методы обратного восстановления также заключаются в восстановлении системы в корректное состояние. Однако процесс восстановления использует информацию о предыдущем полном или частичном состоянии системы, запись о котором называется контрольной точкой. Протоколами восстановления, основанными на контрольных точках, являются: некоординированный протокол (uncoordinated checkpointing); координированный протокол (coordinated checkpointing); протокол вынужденных сообщений (communication-induced checkpointing).

На данный момент наибольшее распространение получил координированный прокол. Однако для него создание контрольных точек связано с большой нагрузкой на сеть, возникающей из-за интенсивного использования узлов ввода/вывода системы для записи на централизованное устройство хранения. Среднее время сохранения контрольных точек может быть весьма значительным и для некоторых систем составляет 20-30 минут [2]. С целью уменьшения затрат на частое создание контрольных точек были разработаны методы, которые включают в себя дополнительные механизмы сохранения, позволяющие уменьшить нагрузку на коммуникационную сеть. Приведем описание трех групп подобных методов.

В первую группу входят методы, основанные на некоординированном протоколе и на механизмах создания журналов передачи сообщений (log-based rollback-recovery). Создание журналов передачи сообщений заключается в том, что на стороне посылающего сохраняется содержание сообщения, информация о получателе и о моменте получения данного сообщения. Использование некоординированного протокола позволяет снизить нагрузку на сеть, так как запись локальных контрольных точек может осуществляться в произвольное время. Во время восстановления, используя журналы передачи сообщений, необходимо произвести дополнительную работу на поиск согласованного состояния системы.

Методы, второй группы [17, 18] основаны на сохранении изменений между двумя последовательными контрольными точками, к ним относятся инкрементный (incremental checkpointing) и дифференциальный (differential checkpointing) методы. Первый предусматривает сохранение в текущей промежуточной контрольной точке изменений относительно предыдущей. Второй сохраняет все изменения относительно базовой контрольной точки. Как показано в работе [18] можно дополнительно использовать методы сжатия для контрольных точек. Восстановление происходит из последней контрольной точки с учетом изменений сохранных в промежуточные контрольные точки.

В третью группу входят методы [15, 19, 20], которые предполагают сохранение контрольных точек вычислительных узлов в память вычислительных узлов. В [15, 19] вводится понятие многоуровневой схемы сохранения контрольных точек, к ним относятся гибридная (hybrid checkpointing) [15] и двухуровневая (two level recovery scheme) [20] схемы. В работе [15] сохранение контрольных точек основано на инкрементном методе с механизмом скрытого копирования. В механизме скрытого копирования после сохранения контрольных точек в память узлов запускается и продолжение расчета, и процесс сохранения согласованной глобальной контрольной точки на централизованное устройство хранения из памяти вычислительных узлов. В работе [19] предлагается использовать метод создания журналов передачи сообщений для сохранения в память вычислительных узлов и координированный протокол для сохранения на централизованное устройство хранения. В [15, 19] при восстановлении используется последняя согласованная глобальная контрольная точка с централизованного устройства хранения и данные из памяти вычислительных узлов. В работе [20] сохранение контрольных точек осуществляется исключительно в память вычислительных узлов (diskless), при этом кроме узлов осуществляющих расчет, необходимы дополнительные узлы для хранения контрольной суммы. В случае отказа память всех узлов используется для реконструкции образа памяти отказавших узлов, затем осуществляется запуск продолжения вычислений.

2.2. Распространенные программные решения

В России наиболее известными программными решениями, в которых реализованы подходы к обеспечению отказоустойчивости, являются следующие разработки: система метакомпьютинга X-Com [21], разработанная сотрудниками НИВЦ МГУ; программный комплекс «Пирамида» [22] разрабатываемый МСЦ РАН совместно с НИИ «Квант»; программная среда «OpenTS»[23], разработанная в Институте программных систем РАН. Данные комплексы обеспечивают работоспособность системы при выходе из строя некоторых вычислительных узлов. Однако X-Com и Пирамида осуществляют организацию вычислительного параллельного процесса исключительно для задач, в которых отсутствуют информационные обмены между обрабатывающими процессами. «OpenTS» – специализированная среда программирования с поддержкой автоматического динамического распараллеливания программ, на основе функционального программирования.

Система управления HTCondor [24] и специальный модуль BLCR для ядра Линукса [25] предоставляют механизмы для сохранения полного контекста процесса и его перезапуска из сохраненного контекста. Эти программные средства являются автоматическими и не позволяют контролировать процедуру сохранения и восстановления. Разрабатываемые в рамках проектов HTCondor и BLCR средства обеспечения отказоустойчивости вычислений не поддерживают в настоящее время отказоустойчивость при выполнении программ, требующих регулярных обменов данными между процессами (например, программ основанных на принципах Domain Decomposition).

В распространенных открытых библиотеках интерфейсов передачи сообщений MPICH [26], MVAPICH [27], OpenMPI [28] реализованы возможности создания контрольных точек по координированному протоколу восстановления на основе специального модуля BLCR для ядра Linux. Таким образом, эти программные средства позволяют обеспечить отказоустойчивость вычислений, требующих регулярных обменов данными между процессами, но сталкиваются с проблемой нагрузки на сеть при сохранении контрольных точек.

Подробный список зарубежных программных решений обеспечивающих отказоустойчивость можно найти в [29].

2.3. Стандарт MPI и расширение ULFM

Стандарт MPI и его программные реализации играют существенную роль в развитии распределенных вычислений. Отметим, что прикладной программист не работает непосредственно с отказами в вычислительных узлах, он сталкивается с их следствием, то есть с отказами в MPI-процессах. Однако в стандарте MPI 3.0 [30] отсутствует описание каких-либо функций связанных с обеспечением отказоустойчивости в вычислительных системах. В работе [1] представлено расширение ULFM [31] стандарта MPI, которое предназначено для решения задач идентификации сбоев в процессе вычислений, восстановления связи между MPI-процессами, исключения из вычислительного поля отказавших MPI-процессов. В качестве модели сбоя принята модель «поломка» устройства вычислительной системы. Данное расширение предоставляет пользователю возможность реализовать в программном приложении различные техники отказоустойчивости, в том числе варьировать объем и содержание контрольных точек. Появление ULMF в стан-

дарте MPI планируется с версии MPI 3.1. Программные реализации MPI 3.1 окажут существенное влияние на возможность проведения расчетов в распределенных системах с частыми отказами.

3. Обеспечение отказоустойчивости на уровне пользователя с помощью локальных контрольных точек

В работе [15] ссылаются на прогноз работы [32]: «... более чем 83% отказов в системах Петафлопсного уровня могут быть восстановлены с использованием локальных контрольных точек, в то время как оставшиеся 17%, включающие сложные ошибки или потерю вычислительного узла, требуют использования доступной согласованной глобальной контрольной точки». Проблема накладных расходов в сети при сохранении контрольных точек и прогноз, приведенный в [32], говорит о необходимости использования памяти вычислительного узла для хранения согласованной глобальной контрольной точки. Реализация методов восстановления на уровне пользователя позволяет значительно сократить объем контрольных точек, по сравнению со стандартным методом [5]. Однако в текущем стандарте MPI 3.0 отсутствуют механизмы позволяющие работать с отказами на уровне пользователя. Несмотря на это, наличие разрабатываемого расширения ULFM [31] для стандарта MPI и его предшественника FT-MPI [33] ведет к необходимости разработки методов автоматического восстановления на уровне пользователя.

3.1. Память вычислительных узлов

Пусть MPI-процессы сохраняют локальные контрольные точки в память своих вычислительных узлов. Тогда, при отказе хотя бы одного вычислительного узла, MPI-процессы, запущенные на других узлах, не смогут получить доступ к локальным контрольным точкам, расположенным в памяти отказавшего. Таким образом, восстановление процесса вычислений будет невозможно. Для выхода и такого положения следует обеспечить избыточность хранения локальных контрольных точек в системе за счет их дублирования в памяти различных вычислительных узлов. Соответствующая схема записи предлагается в следующем параграфе. В её рамках для каждого MPI-процесса определяются номера вычислительных узлов, в память которых должны быть сохранены копии локальной контрольной точки. Загруженность сети при использовании данной схемы сохранения, будет зависеть от объемов локальных контрольных точек.

3.2. Схема сохранения локальных контрольных точек

Для описания схемы записи локальных контрольных точек будем использовать следующие параметры: N – число узлов в системе; MNF – максимально допустимое число отказов в системе; глубина хранения SD (storage depth) – это количество итераций сохранения, в которые каждая локальная контрольная точка доступна для чтения; коэффициент дублирования DF (duplication factor) – это количество узлов хранящих копию данной локальной контрольной точки, отличных от данного вычислительного узла. Цель данной схемы сохранения состоит в максимизации минимального числа узлов, по-

вреждение которых приведёт к невозможности восстановления с последних SD итераций сохранения.

Предполагаем, что на один вычислительный узел приходится один MPI-процесс. Данное предположение не снижает общности последующих рассуждений. При запуске на одном узле нескольких MPI-процессов необходимо всем MPI-процессам с одного узла осуществлять запись и чтение копий локальных контрольных точек в память вычислительных узлов, определяемых формулой (1). В этом случае в системе будет доступна согласованная глобальная контрольная точка, если число отказов узлов не превосходит значение MNF.

Будем предполагать, что используется координированный протокол. В случае использования других методов, формула (2) может быть неверна.

Введем нумерацию итераций сохранения контрольных точек $k = 0,1,2,\ldots$. На одной итерации сохранения MPI-процесс с каждого i–ого вычислительного узла:

- сохраняет свою локальную контрольную точку в память этого узла;
- осуществляет запись DF копий своей локальной контрольной точки в память вычислительных узлов, определяемых формулой (1).

Введем параметр $j \in \{1, 2, \ldots, DF\}$ для обозначения номера передаваемой копии локальной контрольной точки с i–ого узла. Тогда на k-ой итерации сохранения, запись -ой копии с i–ого узла должна быть осуществлена в память вычислительного узла с номером, определяемым формулой:

$$receiver(i,j,k) = \left[i + j \cdot DF^{k \bmod SD} + k \bmod SD\right] \bmod N. \quad (1)$$

В данной схеме объем сохраняемой на каждом узле информации можно оценить величиной $V = V_0 \cdot SD \cdot (DF + 1)$, где V_0 - средний объем локальных контрольных точек.

Пусть в вычислительной системе достаточно много узлов $N \geq DF^{SD} + SD$ и число отказавших узлов не превосходит

$$MNF = (DF - 1) \cdot SD + 1. \quad (2)$$

Тогда, после первых SD итерации сохранения, в системе будут доступны локальные контрольные точки MPI-процессов со всех узлов одной из последних SD итераций сохранения, то есть согласованная глобальная контрольная точка. Таким образом, используя предлагаемую схему, можно восстановить процесс вычислений, если количество отказов в системе не превышает значения MNF.

3.3. Восстановление вычислений после возникновения отказа

Пусть все множество MPI-процессов делится на рабочие процессы, формирующие рабочее поле, и дополнительные процессы. Предполагаем, что в приложении осуществляется координированное сохранение необходимых пользователю данных по описанной выше схеме на вычислительные узлы рабочих процессов. В общем случае, можно осуществлять сохранение в память всех доступных узлов. После возникновения отказа и оповещения системы о нем должны быть выполнены действия по восстановлению:

1. MPI библиотеки и среды выполнения программы;
2. расчетного поля (перераспределение рабочих и дополнительных процессов);
3. промежуточных данных (для процессов в новом рабочем поле);
4. расчета на основе содержимого локальных контрольных точек доставленных на каждый из узлов нового рабочего поля.

Оповещение об отказе в системе, восстановление MPI библиотеки, среды выполнения программы и рабочее поля предполагается осуществлять с помощью стандартных средств, таких как ULFM для MPI. Восстановление расчета на основе данных доставленных на каждый из узлов нового рабочего поля должно осуществляться пользователем.

Код реализующий алгоритм поиска линии восстановления

```
1:  int receiver(int i, int j, int k, int DF, int SD, int N)
2:  {
3:    return (i+j*(int)pow((double)DF,k%SD)+ k%SD)%N;
4:  }
5:  bool is_element_of_Y(int rank, int *fault_nodes, int N_fault){
6:    for(int i = 0; i < N_fault; i++){
7:      if(rank == fault_nodes[i])
8:        return false;
9:    }
10:   return true;
11: }
    int accessibility(int n, int k, int *fault_nodes, int N_fault,int
12: DF, int SD, int N){
13:   int Receiver;
14:   for(int j = 1; j <= DF; j++){
15:     Receiver = receiver(fault_nodes[n], j, k, DF, SD, N);
16:     if(is_element_of_Y(Receiver, fault_nodes, N_fault))
17:       return Receiver;
18:   }
19:   return -1;
20: }
    int RLFFN( int k_end, int *fault_nodes, int N_fault, int
21: *recovery_nodes, int DF, int SD, int N){
22:   int number_of_saving;
23:   for (int k = k_end; k>(k_end-SD); k--){
24:     bool recovery_is_imposible = false;
25:     for(int n = 0; n < N_fault; n++ ){
          if((recovery_nodes[n] = accessibility ( n, k, fault_nodes,
26: N_fault, DF, SD, N))==-1){
27:         recovery_is_imposible = true;
28:         break;
29:       }
30:     }
31:     if(!recovery_is_imposible )
32:       return (number_of_saving = k);
33:   }
34:   return -1;
35: }
```

Перед описанием алгоритма восстановления промежуточных данных введем следующие уточнения. Введем обозначения: U – множество MPI-процессов запускаемого приложения, среди которых $U_{\text{раб}}$ – множество рабочих, $U_{\text{доп}}$ – множество дополнительных процессов. Допустим, что произошли отказы на всех процессах из множества X. Пусть X подмножество $U_{\text{раб}}$, что не ограничивает общность рассуждений, в противном случае удалим из X и $U_{\text{доп}}$ общие процессы. При $|X| \leq |U_{\text{доп}}|$ возможно создание нового расчетного поля $\widehat{U}_{\text{раб}}$ с помощью замены отказавших процессов на нормально функционирующие дополнительные процессы. Пусть Y – множество нормально функционирующих

процессов из начального расчетного поля $U_{\text{раб}}$ (до введения дополнительных процессов) ($Y = U_{\text{раб}} \backslash X$), Z – множество дополнительных процессов, вводимых в новое рабочее поле ($|Z| = |X|$, $\widehat{U}_{\text{раб}} = Y \cup Z$). В этом случае в памяти узлов, соответствующих процессам множества Z, будет отсутствовать информация о каких-либо контрольных точках. Восстановление возможно только из данных сохраненных в памяти узлов, соответствующих процессам множества Y.

Для восстановления промежуточных данных необходимо определить линию восстановления (последнюю согласованную глобальную контрольную точку [5]) и произвести копирование соответствующих локальных контрольных точек из памяти узлов, соответствующих процессам множества Y, в память узлов, соответствующих процессам множества $\widehat{U}_{\text{раб}}$. Предполагаем, что после восстановления MPI библиотеки, среды выполнения и расчетного поля верно следующие:

- номера рабочих процессов из Y совпадают для $U_{\text{раб}}$ и $\widehat{U}_{\text{раб}}$, то есть должны остаться прежними при замене отказавших процессов на нормально функционирующие дополнительные процессы;
- все процессы в новом расчетном поле должны знать параметры восстановления: глубину сохранения контрольной точки, коэффициент дублирования, номер последней итерации сохранения, число отказавших процессов и их номера.

Линия восстановления будет зависеть от того, какую именно согласованную глобальную контрольную точку можно извлечь из памяти узлов, соответствующих процессам множества Y. Для этого требуется:

- определить номер последней доступной согласованной глобальной контрольной точки;
- определить, для каждого процесса из множества $\widehat{U}_{\text{раб}}$, номер процесса из Y, имеющего доступ к соответствующей локальной контрольной точке.

Для линии восстановления каждый процесс из множества Y будет использовать локальную контрольную точку, записанную в память его вычислительного узла. Таким образом, линия восстановления будет определяться номером последней доступной согласованной глобальной контрольной точки и номерами процессов множества Y, имеющих доступ к соответствующим локальным контрольным точкам процессов множества Z.

Определение необходимых для линии восстановления параметров реализовано в функции RLFFN, в коде которой используются следующие переменные и функции:

- k_end – номер последней итерация сохранения;
- N_fault – число отказавших узлов;
- fault_nodes – массив номеров отказавших узлов в прошлом расчетном поле;
- recovery_nodes[i] – номер процесса из Y, который имеет доступ к локальной контрольной точке, необходимой процессу из Z, с номером fault_nodes[i], для линии восстановления;
- number_of_saving – номер последней согласованной глобальной контрольной точки для восстановления;
- bool is_element_of_Y(int rank, int *fault_nodes, int N_fault) – определяет принадлежность данного процесса с рангом rank множеству Y;

- int accessibility(int n, int k, int *fault_nodes, int N_fault,int DF, int SD, int N) – определяет доступность копии локальной контрольной точки процесса fault_nodes[n] от процессов из Y.

Таким образом, алгоритм восстановления промежуточных данных состоит в определении линии восстановления, т.е. запуска на всех MPI-процессах функции RLFFN, и в копировании недостающих локальных контрольных точек согласно линии восстановления. Копирование заключается в том, что процесс с номером recovery_nodes[i] записывает процессу с номером fault_nodes[i] соответствующую локальную контрольную точку, сохраненную на number_of_saving итерации сохранения. После осуществления копирования для всех процессов из Z должна следовать фаза восстановления расчета пользователем.

Заключение

На данный момент остается актуальным развитие методов автоматического восстановления программ, позволяющих снизить нагрузку на сеть во время сохранения контрольных точек. Одно из возможных решений заключается в предоставлении пользователю возможности реализации в его программе различных методов отказоустойчивости, в том числе варьирования объема, содержания и стратегии сохранения контрольных точек. Для реализации методов на уровне пользователя необходима поддержка механизмов работы с отказами, что связанно с внедрением расширения ULFM в стандарт MPI и его реализацией. Привлекательной является стратегия, основанная на использовании для хранения локальных контрольных точек памяти вычислительных узлов. В работе представлена схема сохранения локальных контрольных точек, в рамках которой осуществляется их дублирование, что позволяет восстановить вычислительный процесс, если число отказов не превосходит допустимого числа отказов для данной схемы. Схема может быть использована, в том числе на уровне пользователя, при разработке различных методов автоматического восстановления вычислений, требующих регулярных обменов данными между процессами.

Работа выполнена при поддержке Российского фонда фундаментальных исследований по гранту 13-01-12073 офи_м.

Литература

1. Bland, W. Post-failure recovery of MPI communication capability: Design and rationale / W. Bland, A. Bouteiller, T. Hérault, G. Bosilca, J. Dongarra // International Journal of High Performance Computing Applications. — 2013. — Vol. 27, No. 3. — P. 244–254.
2. Cappello, F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities / Cappello F. // International Journal of High Performance Computing Applications. — 2009. — Vol. 23, No. 3. — P. 212–226.
3. Hsu, C.-H. A power-aware run-time system for high-performance computting / C.-H. Hsu, W.-C. Feng. // Proceedings of SC|05: The ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Seattle, Washington USA November 12 – 18, 2005). — IEEE Press, 2005. — P. 1–9.

4. Sorin, D. Fault Tolerant Computer Architecture. Synthesis Lectures on Computer Architecture / D. Sorin — Morgan&Claypool, 2009. — 104 p.
5. Elnozahy, E.N. A Survey of Rollback-Recovery Protocols in Message-Passing Systems / E.N. Elnozahy, L. Alvisi, Y. Wang, D.B. Johnson // ACM Computing Surveys. — 2002. — Vol.34, No. 3 — P. 375–408.
6. Koren, I. Fault-Tolerant Systems / I. Koren, C. M. Krishna — San Francisco, CA: Morgan Kaufmann Publishers Inc., 2007. — 378 p.
7. Таненбаум, Э. Распределенные системы: принципы и парадигмы / Э. Таненбаум, М. Ван Стеен — Санкт-Петербург: Изд-во Питер, 2003. — 877 с.
8. Kogge, P.M. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems — Tech. Report TR-2008-13. — Univ. of Notre Dame, CSE Dept. — 2008. / P.M. Kogge, et al. URL: http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf (дата обращения: 25.07.2014).
9. Avizienis, A. Basic Concepts and Taxonomy of Dependable and Secure Computing / A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr // IEEE Transactions on Dependable and Secure Computing. — 2004. — Vol. 1, — P. 11–33.
10. Jalote, P. Fault Tolerance in Distributed Systems / P. Jalote — New Jersey, Prentice Hall, 1994 — 448 p.
11. Тель, Ж. Введение в распределенные алгоритмы / Ж. Тель — Москва.: МЦМНО, 2009. — 616 с.
12. The computer failure data repository URL: https://www.usenix.org/cfdr (дата обращения: 25.07.2014).
13. Addressing the challenges of petascale computing for scientific discovery on information storage capacity, performance, concurrency, reliability, availability, and manageability URL: http://pdsi.nersc.gov/ (дата обращения: 25.07.2014).
14. Yuan, Y. Job failures in high performance computing systems: A large-scale empirical study / Y. Yuan, Y. Wu, Q. Wang, G. Yang, W. Zheng // Computers & Mathematics with Applications. — 2012. — Vol. 63, No 2. — P. 365–377.
15. Dong, X. A Case Study of Incremental and Background Hybrid In-Memory Checkpointing / X. Dong, N. Muralimanohar, N.P. Jouppi, Y. Xie // Proceedings of the 2010 Exascale Evaluation and Research Techniques Workshop (Pittsburgh, PA, USA March 13 – 14, 2010), — ACM, 2010 — P. 119–147.
16. Schroeder, B. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? / B. Schroeder, G.A. Gibson // Proceedings of the 5th USENIX Conference on File and Storage Technologies (San Jose, CA, USA February 13–16 2007) — USENIX, 2007. — P. 1–16.
17. Ferreira, K.B. Accelerating incremental checkpointing for extreme-scale computing / K.B. Ferreira, R. Riesen, P.G. Bridges, D. Arnold, R. Brightwell // Future Generation Computer Systems. — 2014. — Vol. 30, No 1. — P. 66–77.
18. Поляков, А.Ю. Оптимизация времени создания и объёма контрольных точек восстановления параллельных программ / А.Ю. Поляков, А.А. Данекина // Вестник СибГУТИ. – Новосибирск: СибГУТИ — 2010. — № 2. — С. 87–100.
19. Vaidya, N.H. A Case for Two-Level Distributed Recovery Schemes / N.H. Vaidya // Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement

and Modeling of Computer Systems (Ottawa, Canada, May 15-19 1995) — ACM, 1995, — P. 64–73.
20. Plank, J.S. Diskless Checkpointing / J.S. Plank, K. Li, M.A, Puening // IEEE Transanctions on Parallel Distributed Systems. — 1998. — Vol. 9, No 10. — P. 972–986.
21. X-COM parallel.ru URL: http://x-com.parallel.ru/node/10 (дата обращения: 25.07.2014).
22. Баранов, А.В. Программный комплекс «Пирамида» организации параллельных вычислений с распараллеливанием по данным / Баранов А.В., Киселёв А.В., Киселёв Е.А., Корнеев В.В., Семёнов Д.В. URL: http://agora.guru.ru/abrau2010/pdf/299.pdf (дата обращения: 25.07.2014).
23. OpenTS - технология и программное обеспечение поддержки распараллеливания программ URL: http://skif.pereslavl.ru/psi-info/rcms-open.ts/index.ru.html (дата обращения: 25.07.2014).
24. HTCondor high throughput computing URL: http://research.cs.wisc.edu/htcondor/index.html (дата обращения: 25.07.2014).
25. Berkeley Lab Checkpoint/Restart (BLCR) for LINUX URL: http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR/ (дата обращения: 25.07.2014).
26. Open MPI: Open Source High Performance Computing URL: http://www.open-mpi.org (дата обращения: 25.07.2014).
27. MPICH URL: http://www.mpich.org (дата обращения: 25.07.2014).
28. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE URL: http://mvapich.cse.ohio-state.edu (дата обращения: 25.07.2014).
29. Egwutuoha, I.P. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. / I.P. Egwutuoha, D. Levy, B. Selic, S. Chen // The Journal of Supercomputing. — 2013. — Vol. 65, No. 3. — P. 1302–1326.
30. Message Passing Interface Forum URL: http://www.mpi-forum.org/ (дата обращения: 25.07.2014).
31. ICL Fault Tolerance URL: http://fault-tolerance.org/ulfm/ulfm-specification (дата обращения: 25.07.2014).
32. Dong, X. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exscale systems, / X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, Y. Xie // Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (Portland, Oregon USA November 14-20, 2009). — ACM, 2009. — P. 57-68.
33. FT-MPI URL: http://icl.cs.utk.edu/ftmpi/people/index.html (дата обращения: 25.07.2014).

Бондаренко Алексей Алексеевич, к.ф.-м.н., научный сотрудник, Институт прикладной математики им. М.В. Келдыша РАН (Москва, Российская Федерация), bondaleksey@gmail.com.

Якобовский Михаил Владимирович, д.ф.-м.н., заведующий сектором «Программное обеспечение вычислительных систем и сетей», Институт прикладной математики им. М.В. Келдыша РАН (Москва, Российская Федерация), lira@imamod.ru

Поступила в редакцию 5 августа 2014 г.

Bulletin of the South Ural State University
Series "Computational Mathematics and Software Engineering"
2014, vol. 3, no. 3, pp. 20–36

FAULT TOLERANCE FOR HPC BY USING LOCAL CHECKPOINTS

A.A. Bondarenko, Keldysh Institute of Applied Mathematics (Moscow, Russian Federation),
M.V. Iakobovski, Keldysh Institute of Applied Mathematics (Moscow, Russian Federation)

One of the main problems that occur in the area of high-performance computing is to continue computations despite of failures. In this paper, we consider the main definitions relating to dependability, briefly review the failure rates for distributed systems and also survey the rollback-recovery approaches. The classic fault-tolerance technique used in parallel applications is the coordinated checkpointing protocol. This protocol takes a consistent global checkpoint snapshot by capturing the local state of each process node simultaneously and saves it on a parallel file system via I/O nodes. However, as the number of compute nodes increases and the size of applications grow, the performance overhead of this protocol can reach an unacceptable level. A solution to this problem is to use local storage for checkpointing. To provide protection, it is necessary to duplicate checkpoints to other local storages. In this work, we develop user level approach and present scheme for checkpointing to the local storages. We proof that, if the number of failures is less than the maximum allowable value for the scheme then it is possible to recover from consistent global checkpoint.

Keywords: parallel computing, fault tolerance, checkpoint, MPI.

References

1. Bland W., Bouteiller A., Hérault T., Bosilca G., Dongarra J. Post-failure recovery of MPI communication capability: Design and rationale. International Journal of High Performance Computing Applications. 2013. Vol. 27, No. 3. P. 244–254.
2. Cappello, F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. International Journal of High Performance Computing Applications. 2009. Vol. 23, No. 3. P. 212–226.
3. Hsu C.-H., Feng W.-C. A power-aware run-time system for high-performance computing. Proceedings of SC|05: The ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Seattle, Washington USA November 12 – 18, 2005). IEEE Press, 2005. P. 1–9.
4. Sorin, D. Fault Tolerant Computer Architecture. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2009. 104 p.
5. Elnozahy E.N., Alvisi L., Wang Y.,Johnson D.B. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys. 2002. Vol.34, No. 3 P. 375–408.
6. Koren I., Krishna C.M. Fault-Tolerant Systems. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2007. 378 p.
7. Tanenbaum A.S., Steen M. Distributed Systems: Principles and Paradigms. New Jersey, Prentice Hall PTR, 2002. 803 p.

8. Kogge P.M. et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Tech. Report TR-2008-13. Univ. of Notre Dame, CSE Dept. 2008. URL: http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf (accessed: 25.07.2014).
9. Avizienis A., Laprie J.C., Randell B., Landwehr C. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing. 2004. Vol. 1, P. 11–33.
10. Jalote P. Fault Tolerance in Distributed Systems. New Jersey, Prentice Hall, 1994. 448 p.
11. Tel G. Introduction to Distributed Algorithms. Cambridge University Press, 2000. 596 p.
12. The computer failure data repository URL: https://www.usenix.org/cfdr (accessed: 25.07.2014).
13. Addressing the challenges of petascale computing for scientific discovery on information storage capacity, performance, concurrency, reliability, availability, and manageability URL: http://pdsi.nersc.gov/ (accessed: 25.07.2014).
14. Yuan Y., Wu Y., Wang Q., Yang G., Zheng W. Job failures in high performance computing systems: A large-scale empirical study. Computers & Mathematics with Applications. 2012. Vol. 63, No 2. P. 365–377.
15. Dong X., Muralimanohar N., Jouppi N.P., Xie Y. A Case Study of Incremental and Background Hybrid In-Memory Checkpointing. Proceedings of the 2010 Exascale Evaluation and Research Techniques Workshop (Pittsburgh, PA, USA March 13 – 14, 2010), ACM, 2010. P. 119–147.
16. Schroeder B., Gibson G.A. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? Proceedings of the 5th USENIX Conference on File and Storage Technologies (San Jose, CA, USA February 13–16 2007) USENIX, 2007. P. 1–16.
17. Ferreira K.B., Riesen R., Bridges P.G., Arnold D., Brightwell R. Accelerating incremental checkpointing for extreme-scale computing. Future Generation Computer Systems. 2014. Vol. 30, No 1. P. 66–77.
18. Polyakov A.Yu. Optimizatsiya vremeni sozdaniya i objema kontrolnykh tochek vosstanovleniya parallelnykh program [Optimization of time creation and checkpoint's volume for parallel programs] // Vestnik SibGUTI [Bulletin of Siberian State University of Telecommunications and Information Sciences]. 2010. No. 2. P. 87–100.
19. Vaidya N.H. A Case for Two-Level Distributed Recovery Schemes. Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (Ottawa, Canada, May 15-19 1995) ACM, 1995. P. 64–73.
20. Plank J.S., Li K., Puening M.A. Diskless Checkpointing. IEEE Transanctions on Parallel Distributed Systems. 1998. Vol. 9, No 10. P. 972–986.
21. X-COM parallel.ru URL: http://x-com.parallel.ru/node/10 (accessed: 25.07.2014).
22. Baranov A.V. Programnyj kompleks «Piramida» organizatsii parallelnykh vychislenij s rasparallelivaniem po dannym [software package «Pyramid» for organization of parallel computing with parallelization of data]. URL: http://agora.guru.ru/abrau2010/pdf/299.pdf (accessed: 25.07.2014).
23. OpenTS – tekhnologiya i programmnoe obespechenie podderzhki rasparallelivaniya programm [Technology and Software Support for Parallelization of Data-Parallel Applica-

tions] URL: http://skif.pereslavl.ru/psi-info/rcms-open.ts/index.ru.html (accessed: 25.07.2014).
24. HTCondor high throughput computing URL: http://research.cs.wisc.edu/htcondor/index.html (accessed: 25.07.2014).
25. Berkeley Lab Checkpoint/Restart (BLCR) for LINUX URL: http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR/ (accessed: 25.07.2014).
26. Open MPI: Open Source High Performance Computing URL: http://www.open-mpi.org (accessed: 25.07.2014).
27. MPICH URL: http://www.mpich.org (accessed: 25.07.2014).
28. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE URL: http://mvapich.cse.ohio-state.edu (accessed: 25.07.2014).
29. Egwutuoha I.P., Levy D., Selic B., Chen S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. The Journal of Supercomputing. 2013. Vol. 65, No. 3. P. 1302–1326.
30. Message Passing Interface Forum URL: http://www.mpi-forum.org/ (accessed: 25.07.2014).
31. ICL Fault Tolerance URL: http://fault-tolerance.org/ulfm/ulfm-specification (accessed: 25.07.2014).
32. Dong X., Muralimanohar N., Jouppi N., Kaufmann R., Xie Y.. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exscale systems. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (Portland, Oregon USA November 14-20, 2009). ACM, 2009. P. 57–68.
33. FT-MPI URL: http://icl.cs.utk.edu/ftmpi/people/index.html (accessed: 25.07.2014).

Received 5 August 2014.