УДК 004.451.24

Оптимизация времени создания и объёма контрольных точек восстановления параллельных программ*)

А.Ю. Поляков, А.А. Данекина

В работе рассмотрены подходы к формированию инкрементных и дифференциальных контрольных точек восстановления параллельных программ. Описан и исследован алгоритм создания контрольных точек, основанный на хешировании, предложена его параллельная версия. Показано, что для итерационных методов решения сложных задач рассмотренные алгоритмы имеют высокую эффективность и позволяют значительно снизить объём ввода-вывода при создании контрольных точек. Оценена эффективность и масштабируемость параллельной версии алгоритма.

Ключевые слова: отказоустойчивость, контрольные точки восстановления программ, распределённые вычислительные системы.

1. Введение

Научно-технический прогресс связан с решением сложных вычислительных задач, что обуславливает потребность в средствах высокопроизводительной обработки информации. Такими средствами являются распределённые вычислительные системы (ВС), основанные на модели коллектива вычислителей [1]. Пиковая производительность современных ВС [2] превышает 1 PetaFLOPS, а количество их вычислительных ядер достигает 300 000.

Среднее время наработки на отказ (Mean time between failure – MTBF) отдельных компонентов распределённых ВС лежит в промежутке от $10^4 - 10^6$ ч. Однако с увеличением количества узлов ВС данный показатель кардинально снижается [3]. На рис. 1 показана зависимость ожидаемого значения MTBF вычислительных систем от количества n и надёжности составляющих узлов.

В табл. 1 приведены статистические данные (количество ядер, MTBF) для нескольких большемасштабных вычислительных систем [4 – 6].

Таблица 1

Вычислительная система	Количество ядер	MTBF		Год создания
		ч.	дни	
IBM's Blue Gene/P	294912	168	7	2008
IBM's Blue Gene/L	131072	148	6.3	2002
Jaguar@ORNL	23.416	37.5	1.5	2003
ASCI White	8192	40	2	2003
ASCI Q	8192	6.5	0.27	2003
Google	15000	1	0.05	2003

*) Работа выполнена при поддержке Совета по грантам Президента РФ (ведущая научная школа НШ 5176.2010.9) и Российского фонда фундаментальных исследований (гранты 09-07-00095, 08-07-00022).

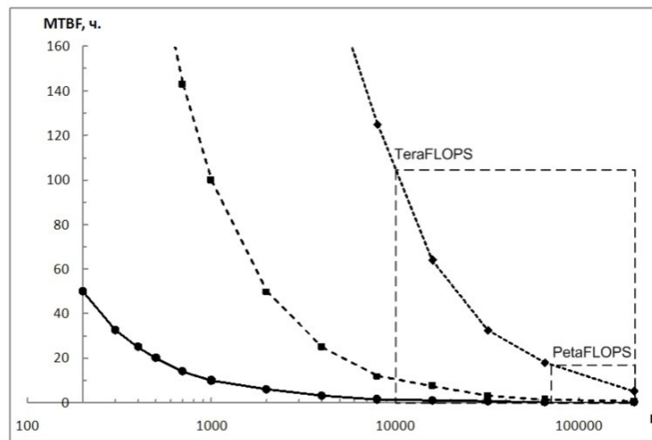

Рис. 1. Зависимость среднего времени наработки на отказ ВС:
MTBF отдельных компонентов: – 10^4 ч. – 10^5 ч. – 10^6 ч.

Существует множество сложных научных, инженерных и экономических задач, для решения которых даже на современных ВС требуется несколько недель. При этом количество таких задач постоянно растёт. Учитывая приведённую выше статистику отказов в большемасштабных ВС, вероятность потери результатов вычислений очень высока.

Таким образом, существует серьёзная необходимость обеспечения отказоустойчивого выполнения программ на ВС. Выделяют два основных подхода к решению данной проблемы [7]: *реактивный* (*reactive*) и *проактивный* (*proactive*). Наибольшее распространение получил реактивный подход, также называемый *Checkpoint/Restart* или *Rollback/Recovery* [8]. Он предусматривает периодическое создание *контрольных точек* (КТ) восстановления, хранящих состояние выполняющейся программы. В случае отказа одного или нескольких вычислительных узлов (ВУ) любая КТ может быть использована для повторного запуска программы на исправной подсистеме. При этом работа продолжится с момента времени, соответствующего созданию этой КТ. Реактивный подход используется также для балансировки нагрузки ВС [9] и в воспроизводящих отладчиках (Playback Debuggers [10, 11]).

Применение механизма КТ связано с накладными расходами при выполнении параллельных программ (ПП). Данная операция характерна интенсивным использованием узлов ввода-вывода (УВВ), поэтому среднее время создания КТ может быть весьма значительным [12]. В работе рассматривается задача создания *инкрементных* и *дифференциальных* КТ, которые содержат только изменённые фрагменты (дельты). Предложена параллельная версия алгоритма формирования инкрементных и дифференциальных контрольных точек, названная нами *P-FHash*.

2. Средства создания контрольных точек восстановления программ

Существует достаточно много средств создания контрольных точек (ССКТ) [13 – 15], каждое из которых имеет свои преимущества и недостатки. Рассмотрим несколько подходов к классификации ССКТ.

Различают две основные схемы взаимодействия ССКТ с защищаемой программой: *явная* и *прозрачная* (*неявная*). Средства создания КТ, построенные на основе явной схемы, позволяют задать ограниченный набор информации, которую необходимо сохранить в КТ. Это обеспечивает малые объёмы дискового ввода-вывода информации и значительно уменьшает накладные расходы таких ССКТ. Недостатком явной схемы является необходимость модификации исходного кода, что не позволяет применять её к программам, доступным только в бинарном виде. Кроме того, КТ могут создаваться только в моменты времени, определяемые программой и связанные с завершённостью определённого этапа вычислений.

Средства создания КТ, построенные на основе прозрачной схемы, осуществляют сохранение КТ незаметно для программы, что обеспечивает простоту и универсальность их использования. Недостатком данного подхода является больший объём дискового ввода-вывода информации, так как сохраняется всё пространство памяти.

По классам поддерживаемых программ ССКТ можно разделить на *сосредоточенные* и *распределённые*. Сосредоточенные ССКТ обеспечивают отказоустойчивость выполнения одного или нескольких процессов в рамках вычислительного узла ВС. Распределённые ССКТ обычно строятся на базе сосредоточенных и применимы для распределённых и параллельных программ, что делает их важным инструментом организации функционирования ВС. Для создания распределённых контрольных точек (РКТ) необходимо:

1) сформировать сосредоточенные КТ для всех процессов, входящих в состав распределённой программы (РП);
2) сохранить граф связей между процессами РП;
3) сохранить сообщения, которые были отправлены, но не доставлены на момент создания РКТ (такие сообщения также называют *транзитными – in-transit*).

Для распределённых ССКТ различают *координированный* и *некоординированный* подходы. При создании РКТ каждый процесс РП сохраняет свое состояние в КТ. Целостной РКТ [8] называется набор из N локальных КТ, формирующих допустимое состояние программы. Такая РКТ может быть использована для восстановления программы после сбоя. При координированном подходе создание КТ происходит синхронно, что гарантирует целостность РКТ. При некоординированном подходе каждый процесс создает КТ независимо от других. Следовательно, при восстановлении необходимо выполнять поиск целостного состояния программы на основе набора независимых КТ. Последнее вносит дополнительные накладные расходы. Для некоординированного подхода существует опасность возникновения «эффекта домино», когда в процессе поиска целостного состояния происходит откат к началу программы.

Распределённые ССКТ также можно разделить на *универсальные* и *MPI-ориентированные*. Первые ССКТ [13] позволяют создавать РКТ для любых распределённых и параллельных программ, в том числе для различных реализаций модели передачи сообщений (PVM, MPI). Что касается вторых, то существует несколько ССКТ, построенных на базе конкретных реализаций MPI (например, OpenMPI [14], MVAPICH2 [15]). Все они используют пакет BLCR [16] для создания сосредоточенных КТ и реализуют собственные механизмы сохранения графа связей и транзитных сообщений.

При создании КТ процесса необходимо сохранить информацию о его состоянии. Это может быть реализовано на нескольких программных уровнях.

1. Операционная система (ОС). Предусматривается сохранение содержимого пространства ядра и пространства пользователя для всех процессов ОС. Такой подход может быть реализован с использованием систем виртуализации (например, VMWare).

2. Ядро ОС. Требуется внедрение дополнительных компонентов, позволяющих сохранить необходимую информацию: содержимое памяти конкретного процесса и информацию о состоянии ядра, относящуюся к нему.

3. Системные библиотеки. Предусматривается сохранение содержимого памяти и состояния ядра с использованием средств, предоставляемых ОС для управления процессами.

4. Прикладной уровень. Предопределяется сохранение минимального объёма информации, необходимого для восстановления каждой конкретной программы.

Средства создания КТ уровней ОС, ядра и системных библиотек реализуются в рамках прозрачной схемы. Кроме того, некоторые ССКТ уровней ядра и системных библиотек предоставляют программе возможность влиять на процесс обеспечения отказоустойчивости, например, выбирать наиболее удобные моменты для создания КТ. Прикладной уровень предусматривает только явную схему.

Преимуществом первого уровня является простота реализации, а недостатком – значительный объём дискового ввода-вывода информации и отсутствие гибкости. Второй уровень позволяет получать прямой доступ к внутренним структурам ядра и памяти процесса и выполнять сохранение необходимой для восстановления информации при меньшем объёме ввода-вывода. Недостатком данного подхода является зависимость от изменений в ядре ОС (новые версии ядра Linux выходят в среднем с частотой раз в 3 – 4 месяца). Также данный подход требует привилегий суперпользователя для установки и управления, а ошибки, допущенные в программном обеспечении уровня ядра, могут приводить к нарушению работы всей ОС.

Третий уровень позволяет обеспечить создание КТ, не требуя при этом привилегий суперпользователя и не подвергая угрозе функционирование всей ОС. Однако при данном подходе невозможно осуществить прямой доступ к внутренним структурам ядра, которые описывают защищаемый процесс. Для этого требуется перехват и обработка системных вызовов. На четвёртом уровне сохраняется лишь содержимое буферов, которые явно указываются в программе.

Ниже рассмотрен универсальный подход, позволяющий снизить накладные расходы любых рассмотренных ССКТ.

3. Инкрементные и дифференциальные КТ

3.1. Проблема накладных расходов ССКТ

Как показано на рис. 1, с увеличением количества ВУ среднее время наработки на отказ снижается. Так, ВС с производительностью порядка ExaFLOPS будут иметь значительно меньшее среднее время между отказами, чем современные PetaFLOPS-системы. По некоторым прогнозам, для таких ВС значение MTBF будет менее получаса. В связи с этим возникает необходимость уменьшать интервал между созданиями КТ.

Количество УВВ в современных ВС адекватно масштабируются вместе с изменением числа ВУ. На практике выдерживается соотношение 32 – 128 ВУ на один УВВ. Например, система IBM BlueGene/L позволяет устанавливать один УВВ на 8 ВУ, но реально это соотношение составляет 1/64 [5] или 1/128 [17]. В табл. 2, согласно данным [12] Ливерморской национальной лаборатории, показана статистика времени создания КТ для нескольких ВС из списка TOP500 [2]. Из таблицы видно, что среднее время создания КТ составляет 20 – 30 мин.

Таблица 2

Вычислительная система	Пиковая производительность, TeraFLOPS	Среднее время создания КТ, мин.
LANL RoadRunner	1000	~20
LLNL BlueGene/L	500	20
Argonne BlueGene/P	500	30
LLNL Zeus	11	26

Далее будет рассмотрен подход к оптимизации создания КТ восстановления программ, предусматривающий формирование инкрементных и дифференциальных КТ. В сочетании с масштабируемостью УВВ позволяет существенно снижать накладные расходы.

3.2. Основные понятия

Как было отмечено ранее, КТ содержит состояние выполняющейся параллельной программы. Оно включает в себя содержимое памяти, информацию об используемых ресурсах

ОС, граф информационных связей и транзитные сообщения. При этом для многих параллельных программ характерно частичное изменение содержимого памяти. Таким образом, существует возможность применения методики двоичного сжатия [18] с помощью дельт (дельта-сжатия). При таком подходе выбирается базовая КТ, которая сохраняется на УВВ полностью. Для всех остальных КТ сохраняются лишь модифицированные фрагменты. Существует два варианта дельта-сжатия: инкрементное и дифференциальное. Первый подход предусматривает сохранение изменений в текущей КТ относительно предыдущей (рис. 2, а). Второй – сохранение всех изменений относительно базовой КТ (рис. 2, б).

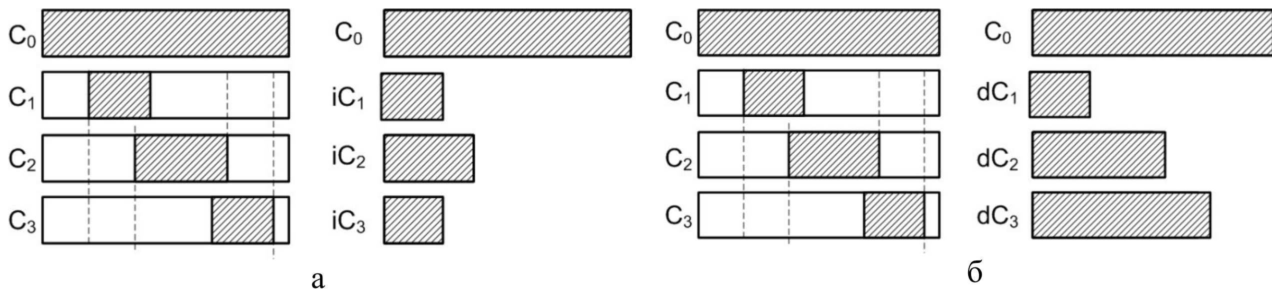

Рис. 2. Контрольные точки:
а – инкрементные; б – дифференциальные;
– фрагменты, которые сохраняются на УВВ;
– фрагменты, не сохраняемые на УВВ

Введем следующие обозначения: C_i – КТ с порядковым номером i, iC_i – i-я инкрементная КТ (ИКТ), dC_i – i-я дифференциальная КТ (ДКТ), xC_i – i-я ИКТ или ДКТ. Функция $S(X)$ определяет размер X в байтах. Функция $inc(C_{i-1}, C_i)$ формирует ИКТ iC_i, функция $dif(C_0, C_i)$ – ДКТ dC_i. При этом справедливо: $S(iC_i) \le S(dC_i) \le S(C_i)$.

Преимуществом дифференциального дельта-сжатия является быстрый алгоритм получения результирующей КТ C_i, Для этого требуется базовая КТ C_0 и дифференциальная КТ dC_i:

$$C_i = dif^{-1}(C_0, dC_i).$$

Недостатком является постоянное увеличение размера дифференциальной КТ (ДКТ):

$$S(dC_1) \le S(dC_2) \le \ldots \le S(dC_i).$$

Инкрементные КТ более эффективны с точки зрения объёма. Однако для получения результирующей КТ C_i требуются базовая C_0 и инкрементные с первой по текущую ($iC_1,...,iC_i$):

$$C_i = inc^{-1}(C_0, iC_1, \cdots, iC_i).$$

Существуют два подхода к дельта-сжатию: страничный [19, 20] и основанный на хешировании [17, 21]. Первый требует аппаратной поддержки в процессоре и программной в ОС и ССКТ. Недостатком данного подхода является отсутствие гибкости при выборе минимальной единицы модификации, которая имеет фиксированный объём, равный размеру страницы памяти. Так, изменение даже одного байта приведёт к сохранению страницы целиком. При этом уменьшение размера страницы приводит к увеличению накладных расходов на управление виртуальной памятью. Второй недостаток – это требования, накладываемые на процессор. В высокопроизводительных системах наблюдается тенденция к упрощению ВУ для повышения надёжности и производительности. Так, в процессорах ВУ системы BlueGene/L аппаратная поддержка защиты памяти отключена.

Методы, основанные на хешировании, используют хеш-функции для обнаружения изменённых фрагментов КТ. Существует также возможность варьирования точности алгоритма и скорости его работы выбором размера минимальной единицы модификации (блока). Этот подход свободен от аппаратных ограничений, накладываемых на предыдущую технологию. Он может быть использован как непосредственно в ССКТ, так и в независимых программных средствах, например, в файловых системах, ориентированных на хранение КТ [21], и системах резервного копирования [22].

4. Алгоритмы формирования инкрементных и дифференциальных контрольных точек

4.1. Алгоритм FHash

На рис. 3 показана схема применения алгоритма, названного нами FHash, который является распространённым подходом к формированию ИКТ и ДКТ [17, 21]. Он не требует для своей работы наличия исходной КТ, вместо этого используется её *сигнатура* (*отпечаток, снимок*) g_i. При этом g_i много меньше C_i ($S(g_i) << S(C_i)$) и может храниться в оперативной памяти (ОП) ВУ.

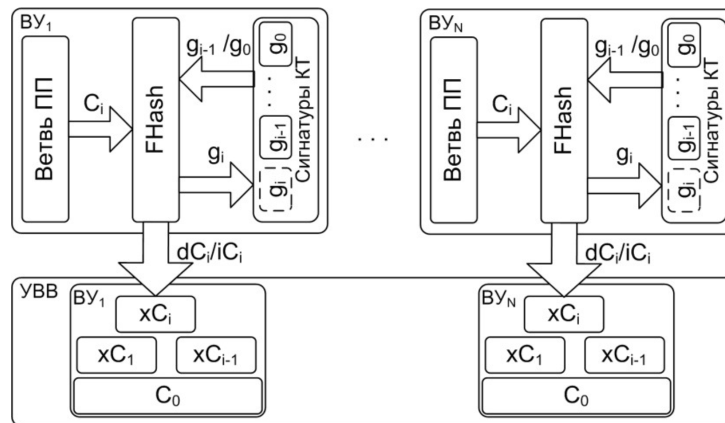

Рис. 3. Схема применения алгоритма FHash

Для вычисления сигнатуры g_i контрольная точка C_i разбивается на K блоков длиной l байт (за исключением, быть может, последнего), где $K =]S(C_i)/l[$ – ближайшее к $S(C_i)/l$ целое, такое что $K \geq S(C_i)/l$. Для каждого блока C_{ik} вычисляется хеш-функция h, на базе которой формируется алгоритм. Таким образом, сигнатура g_i представляет собой совокупность хеш-кодов всех блоков C_i:

$$g_i = \{h(C_{i1}), \cdots h(C_{iK})\}. \quad (1)$$

Рассмотрим процесс дельта-сжатия с использованием алгоритма FHash. На вход FHash поступает очередная КТ C_i и сигнатура ($\tilde{g}$), относительно которой выполняется сжатие (для ИКТ – это g_{i-1}, для ДКТ – g_0). Далее по формуле (1) вычисляется сигнатура g_i и на основе сравнения g_i и $\tilde{g}$ определяются блоки, которые должны быть сохранены в формируемой ИКТ (ДКТ) xC_i:

$$xC_i = \{C_{ik} \mid g_{ik} \neq \tilde{g}_{ik}, k = 1,...,K\}$$

4.3. Алгоритм P-FHash

Алгоритм FHash характеризуется высокой скоростью работы [22, 23], тем не менее, он создает дополнительные накладные расходы в процессе выполнения программ. Наиболее трудоёмким шагом FHash является формирование сигнатуры, поскольку он требует найти K значений функции h. В качестве h обычно выбирается «сильная» хеш-функция с малой вероятностью коллизии (например, MD4[24], MD5[25], SHA1[26]), вычисление которой требует значительного процессорного времени.

Как было сказано ранее, наиболее распространённым на данный момент является координированный подход к созданию КТ. В этом случае во время создания КТ выполнение всех ветвей параллельной программы приостанавливается. Современные ВС обычно формируются из ВУ, которые объединяют несколько процессоров, состоящих, в свою очередь, из нескольких вычислительных ядер (ВЯ) [5, 6]. Поэтому на ВУ может одновременно выполняться несколько ветвей одной параллельной программы. Таким образом, в момент создания КТ программе формирования ИКТ (ДКТ) может быть доступно более одного ВЯ. Поэтому актуальным является повышение производительности алгоритма за счёт использования параллелизма ВУ.

Алгоритм вычисления сигнатуры состоит из ряда независимых шагов и может быть эффективно распараллелен. В данной работе была использована методика крупноблочного распараллеливания [1], которая характеризуется высокой эффективностью и низкими накладными расходами.

В связи с тем, что в момент создания КТ оперативная память (ОП) вычислительного узла содержит данные и код ПП, объём ОП, которую может использовать FHash, ограничен. Данное требование обусловлено несколькими факторами.

1. Так как долговременная память имеет низкую производительность, необходимо не допускать выгрузку страниц памяти, принадлежащих ПП, на постоянное запоминающее устройство (ПЗУ).

2. Существуют также конструктивные ограничения, например, ВУ системы IBM BlueGene не имеют локальных ПЗУ.

В связи с приведёнными выше причинами алгоритм P-FHash обрабатывает КТ фрагментами. На рис. 4 показана схема распределения данных по вычислительным ядрам ВУ.

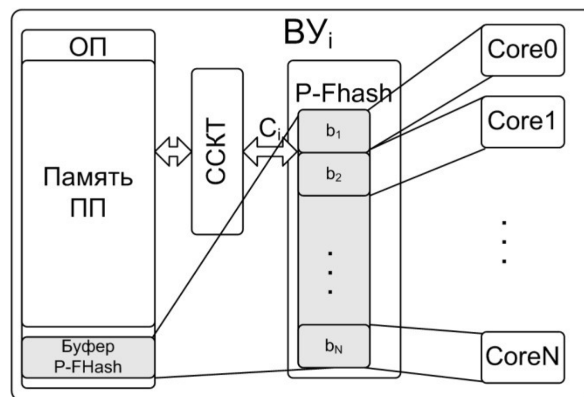

Рис. 4. Распределение данных по вычислительным ядрам

На вход параллельного алгоритма передаётся количество ВЯ N, сигнатура $\tilde{g}$, контрольная точка C_i, которая поступает фрагментами через буфер b, и размер буфера в блоках – m. Далее выполняется распределение данных по ВЯ в соответствии с приведённым алгоритмом:

for $n := 1$ **to** N **do {**
$m_n := [m / N]$
if $n < m \bmod N$ **then** { $m := m + 1$ }
}

Далее выполняется вычисление сигнатуры в соответствии со следующим алгоритмом:
$p := 0$
while $p < K$ **do{**
$b = < C_{ip}, C_{i,p+1}, ..., C_{i,p+m} >$
$ComputeHash(N, b, p, m, m_1, ..., m_N)$**;**
for $n := 1$ **to** N **do**{
$k := p + \sum_{z=1}^{n-1} m_z$
for $j := 1$ **to** m_n **do**{
if $\tilde{g}_{k+j} \neq g_{nj}$ **then**{
$SaveBlock(C_{i,k+j})$
}
}
$p := p + m$
}

Функция *ComputeHash* выполняет параллельное вычисление сигнатуры для указанного фрагмента. Алгоритм работы ВЯ с номером *n* представлен ниже:

for $r := p + \sum_{z=1}^{n-1} m_z$ **to** $p + \sum_{z=1}^{n-1} m_z + m_n$ **do{**
$g_{nr} = h(b_r)$
}

Блоки, которые были изменены, последовательно записываются в ИКТ(ДКТ) с использованием функции *SaveBlock*.

5. Моделирование алгоритмов формирования инкрементных и дифференциальных КТ

5.1. Реализация алгоритмов FHash и P-FHash

Алгоритмы FHash и P-FHash реализованы на языке программирования C++. В качестве h использовалась хеш-функция MD5, имеющая низкую вероятность коллизии (10^{-19}). Также предусмотрен интерфейс, позволяющий изменять используемую функцию h.

Для создания КТ использовались пакеты BLCR [16] и DMTCP [13]. Алгоритмы не учитывают структуру КТ, она рассматривается как набор битов. Это позволяет применять созданные средства для КТ произвольных ССКТ. Параллельная версия разработана с использованием технологии OpenMP.

Схема применения разработанных программных компонентов показана на рис. 5. Поток данных, содержащий КТ, направляется через конвейер программе, реализующей описанные алгоритмы. На основе сигнатуры, которая также является входными данными, выполняется обработка КТ и полученная ИКТ (ДКТ) записывается непосредственно в файл или дополнительно сжимается.

5.2. Выбор реализации хеш-функции

Как уже было сказано ранее, эффективность работы алгоритма FHash существенно зависит от времени вычисления функции h. В качестве h нами была выбрана хеш-функция MD5 и проведено сравнение её существующих реализаций. Рассматривались следующие три: Стокса, Ривеста (автора MD5) и openSSL. На рис. 6 представлена зависимость времени работы хеш-функции t от размера хешируемого блока S. На графике видно, что при S более 500 КБ реализация библиотеки openSSL примерно в 2 раза эффективнее вариантов Стокса и Ривеста, которые имеют одинаковую производительность. По этой причине в дальнейших исследованиях использовалась библиотека openSSL.

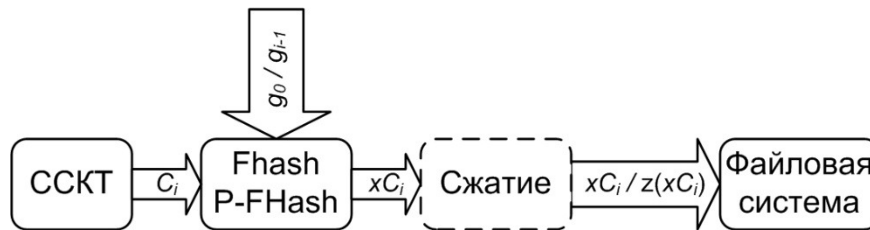

Рис. 5. Формирование ИКТ

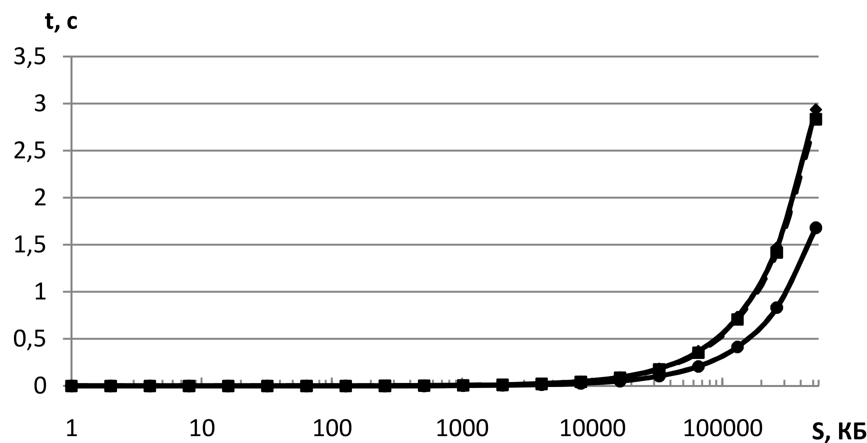

Рис. 6. Реализация алгоритма MD5:
—●— – OpenSSL; - ■ -– Стокса; --◆-- – Ривеста

5.3. Анализ эффективности алгоритма FHash

Эффективность алгоритма FHash была исследована для программ, выполняющих решение систем линейных алгебраических уравнений (СЛАУ). Рассматривался точный метод решения СЛАУ с помощью LU-разложения, реализованный в наборе тестов NPB (NAS Parallel Benchmarck [27]). КТ создавались с помощью пакетов BLCR и DMTCP для классов задач B и C. На рис. 7.а показано отношение размера созданной ИКТ к размеру исходной КТ. В среднем объём ИКТ составил 65 – 70 %.

Сходные результаты были получены коллективом исследователей из компании IBM [17] алгоритмом, аналогичным FHash. Для задачи LU-разложения матриц процент различий составил 55 %. Различие объясняется тем, что в работе [17] учитывалась структура КТ.

Наряду с точными методами решения СЛАУ нами были рассмотрены итерационные методы. В качестве тестовой программы использовались реализации итерационных методов из библиотеки IML++ [28]. Все КТ создавались с помощью ССКТ DMTCP. На рис. 7 показаны результаты, полученные для различных методов.

Для метода бисопряженных градиентов (рис. 7, б) все ИКТ, кроме первой, имеют низкий процент отличий от предыдущей КТ (менее 2 %). Больший размер первой ИКТ объясняется

тем, что между первой и второй КТ вычислялась матрица предобуславливания и, следовательно, значительно изменялось содержимое памяти. Схожую ситуацию можно наблюдать и для методов обобщённых минимальных невязок (рис. 7, в) и псевдоминимальных невязок (рис. 7, г).

Таким образом, для различных программ формирование ИКТ имеет разную эффективность, которая в значительной мере зависит от шаблона использования памяти. Если выполняется обработка больших массивов данных и формируется результаты, имеющие значительно меньший объём, то подход, использующий ИКТ, будет иметь высокую эффективность. В противном случае размеры ИКТ будут близки к размерам исходных.

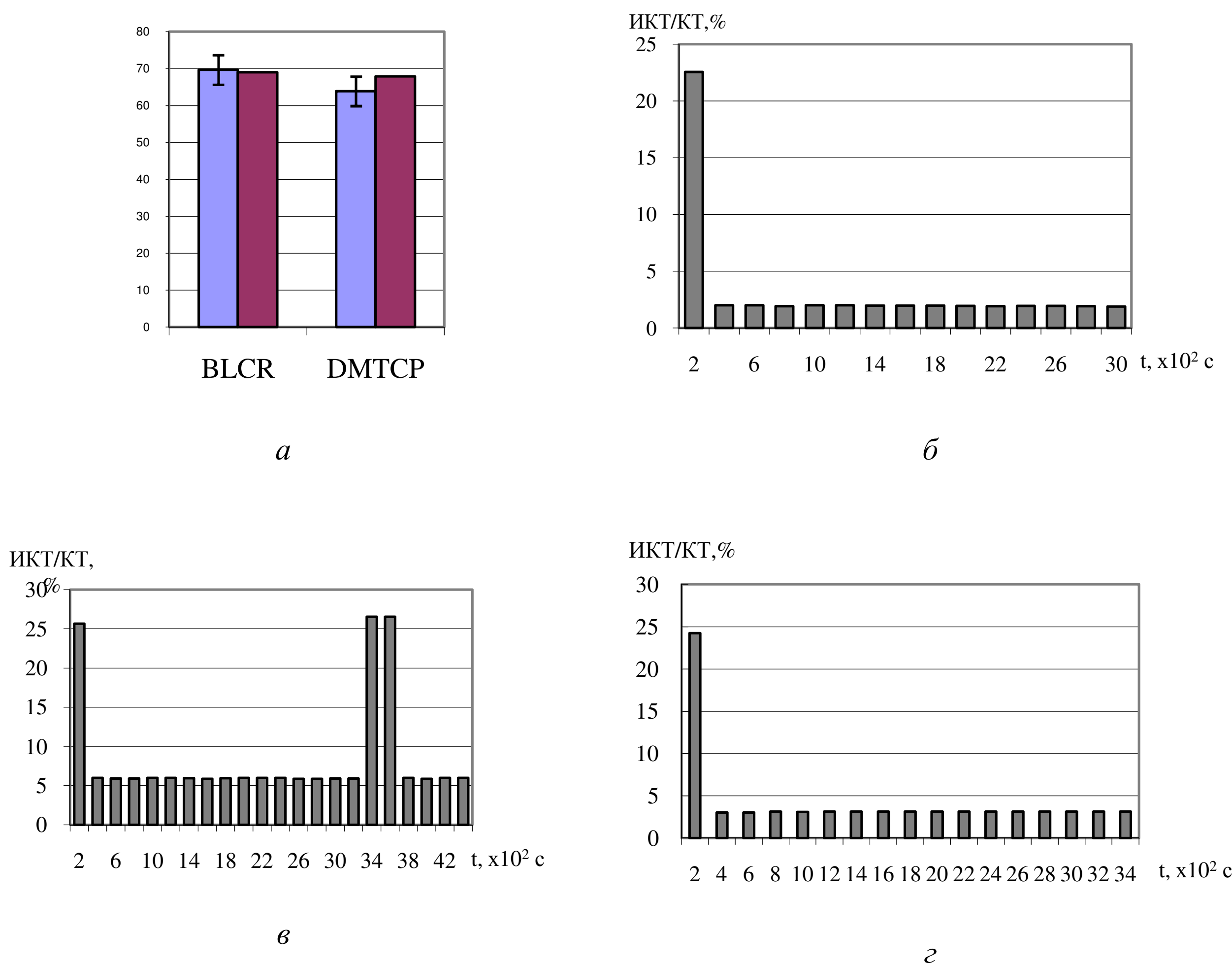

Рис. 7. Формирование ИКТ для программ решения СЛАУ:
а – метод LU разложения, NPB; *б* – Метод бисопряжённых градиентов, IML
в – Метод обобщённых минимальных невязок, IML; *г* – Метод псевдоминимальных невязок, IML

5.4. Моделирование алгоритма P-FHash

Исследование масштабируемости предложенного параллельного алгоритма P-FHash производилось на двух типах ВУ (ВУ$_1$, ВУ$_2$), характеристики которых приведены в табл. 3.

На рис. 8 показан график ускорения алгоритма на ВУ$_1$. Ухудшение эффективности на 15 и 16 ВЯ объясняется тем, что ВУ$_1$ не был в эксклюзивном пользовании и на момент замеров имел загрузку, приблизительно равную 1.5 ВЯ. Ветви, распределявшиеся на частично загруженные ВЯ, выполнялись медленнее, замедляя все остальные. Замеры проводились для КТ размером 3.8 ГБ и 28 ГБ, буфер алгоритма имел размер $S(B) = 400$ МБ на ВЯ. Размер блока $l = 1$ КБ.

На рис. 9 показан график ускорения алгоритма на ВУ$_2$. Данный узел находился в эксклюзивном использовании. Поэтому наблюдается постоянный прирост производительности. Замеры проводились для КТ размером 3.8 ГБ, буфер алгоритма имел размер $S(B) = 50$ МБ на ВЯ. Размер блока (l) также составлял 1 КБ.

Таблица 3

Характеристика	**ВУ$_1$**	**ВУ$_2$**
Количество процессоров	4	2
Количество ВЯ	16	8
Тип процессора	AMD Quad-Core Opteron 8346 HE	Intel Quad-Core Xeon E5420
Количество ВЯ на процессор	4	4
Частота	1.8 ГГц	2.5 ГГц
Кеш L2	4×512 КБ	2×6 МБ
Кеш L3	2 МБ	-
Объём ОЗУ	126 ГБ	8 ГБ

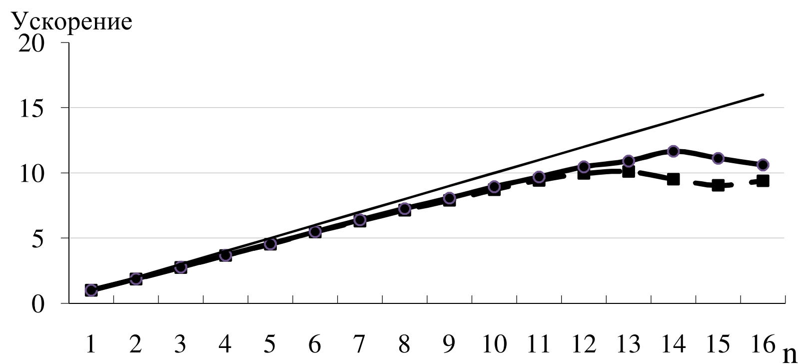

Рис. 8. Ускорение на ВУ$_1$:
– размер КТ 28 ГБ; – размер КТ 3.8 ГБ

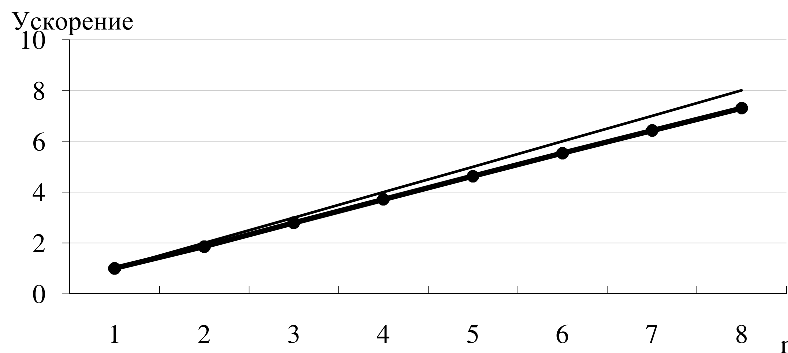

Рис. 9. Ускорение на ВУ$_2$
– размер КТ 3.8 ГБ;

6. Заключение

В работе предложена параллельная версия алгоритма создания инкрементных и дифференциальных контрольных точек восстановления программ. Показана эффективность данного подхода. Установлено, что для итерационных методов, для которых характерно незначительное изменение хранимых данных, размер контрольной точки может быть уменьшен на порядок. Например, для реализации алгоритмов решения систем линейных алгебраических

уравнений размер инкрементной контрольной точки составил 2 – 6 % от размера исходной. Применение подхода для некоторых точных методов менее эффективно, но также целесообразно: для реализации решения систем линейных алгебраических уравнений методом LU-разложения размер инкрементной контрольной точки составил 70 % от размера исходной.

Построенный график для коэффициента ускорения параллельной версии алгоритма показывает его масштабируемость на различных аппаратных платформах. В частности, при обработке контрольной точки размером 28 Гигабайт на 14 вычислительных ядрах требуется 12.72 сек., в то время как на одном – 130 сек.

Предложенный алгоритм реализован в пакете HBICT (Hash based incremental checkpointing [29]), который создан и развивается в Центре параллельных вычислительных технологий ГОУ ВПО «СибГУТИ».

Литература

1. Хорошевский В.Г. Архитектура вычислительных систем. – М.: МГТУ им. Н.Э. Баумана, 2008 . 520 с.
2. TOP500 Supercomputer sites [Электронный ресурс]. URL: http://www.top500.org/
3. Fabrizio Petrini and Kei Davis and José Carlos Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. In In 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems (FTPDS04), Santa Fe, NM, April 2004.
4. Ian Philp. Software failures and the road to a petaflop machine. In HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11). IEEE Computer Society, 2005.
5. T. B. Team. An overview of the BlueGene/L supercomputer. In Proceedings of SC2002: High Performance Networking and Computing, Baltimore, MD, Nov. 2002.
6. Tom Budnik, Brant Knudson, Mark Megerian, Sam Miller. High Throughput Computing on IBM's Blue Gene®/P // IBM Rochester Blue Gene Development. – Режим доступа: http://www-03.ibm.com/systems/resources/HTC_WhitePaper_V2_050508.pdf (дата обращения: 01.06.2010)
7. A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07), Seattle, WA, June 2007.
8. Elnozahy E.N., Alvisi L., Wang Y.M., Johnson D.B. A survey of rollback-recovery protocols in message-passing systems // ACM Computing Surveys. – Vol. 34, No 3, 2002. – pp. 375-408.
9. C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA, Nov. 2008
10. Ana Maria Visan, Artem Polyakov, Praveen S. Solanki, Kapil Arya, Tyler Denniston, Gene Cooperman Temporal Debugging using URDB, 2009. – Режим доступа: http://arxiv.org/abs/0910.5046v1 (дата обращения: 01.06.2010).
11. Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung and Chandra Kintala. «Checkpointing and Its Applications», 25th Annual Int'l symposium on Fault-Tolerant Computing, PP. 22 – 30, Oct. 1995.
12. Xiangyu Dong, Naveen Muralimanohar, Norman P. Jouppi, Yuan Xie. A Case Study of Incremental and Background Hybrid In-Memory Checkpointing, The Exascale Evaluation and Research Techniques Workshop (EXERT) at ASPLOS 2010, March 2010.

13. J. Ansel, K. Arya, G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop // Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS'09). IEEE Press, 2009.
14. J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE Computer Society, March 2007.
15. Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. Parallel Processing, Jan 2006.
16. Paul H. Hargrove and Jason C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In Proceedings of SciDAC 2006: June 2006.
17. S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In ICS 2004, pages 277 – 286, 2004.
18. R. C. Burns and D. D. E. Long, «Efficient distributed backup with delta compression». In Proceedings of the 1997 I/O in Parallel and Distributed Systems (IOPADS’97), San Jose, CA, USA, Nov. 1997.
19. Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, Kei Davis. Transparent, Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers // In Proceedings of the 2005 International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing 2005), Seattle, WA, November 2005
20. Sangho Yi, Junyoung Heo, Yookun Cho, Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time // Proc. of the 2006 ACM symposium on Applied computing table of contents Dijon, France, pp. 1472 – 1476, 2006
21. S.A. Kiswany, M. Ripeanu, S. S. Vazhkudai, A. Gharaibeh, «stdchk: A Checkpoint Storage System for Desktop Grid Computing», Proc. of ICDCS 2008, Beijing, China, 2008.
22. Rsync [Электронный ресурс]. URL: `http://rsync.samba.org/`. (дата обращения 14.05.2010).
23. Muthitacharoen, A., B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In Symposium on Operating Systems Principles (SOSP). 2001. Banff, Canada.
24. Rivest, R., The MD4 message digest algorithm // Proc. of Advances in Cryptology - CRYPTO'90, pp. 303 – 311, Springer-Verlag, 1991.
25. Rivest, R., The MD5 Message-Digest Algorithm, RFC1321, April 1992.
26. FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.
27. D. Bailey, T. Harris, W. Saphir, R. vander Wijngaart, A. Woo, and M. Yarros, «The NAS parallel benchmarks 2.0,» Tech. Rep. NAS-95-020, NAS Systems Division, Dec. 1995.
28. IML++ (Iterative Methods Library) [Электронный ресурс]. URL: `http://math.nist.gov/iml` (дата обращения 14.05.2010)
29. HBICT (Hash based incremental checkpointing tool) [Электронный ресурс]. URL: `http://sourceforge.net/projects/hbict/` (дата обращения 06.06.2010)

Статья поступила в редакцию 15.05.2010

Поляков Артём Юрьевич

Младший научный сотрудник Лаборатории вычислительных систем Института физики полупроводников им. А.В. Ржанова СО РАН, старший преподаватель Кафедры вычислительных систем Сибирского государственного университета телекоммуникаций и информатики. Область научных исследований – отказоустойчивое выполнение параллельных программ, самоконтроль и самодиагностика ВС.

Тел.&факс (383) 333-21-71, 269-82-75, e-mail: artpol84@gmail.com.

Данекина Анна Александровна

Магистрант Кафедры вычислительных систем Сибирского государственного университета телекоммуникаций и информатики. Область научных исследований – отказоустойчивое выполнение параллельных программ.

Тел.&факс (383) 333-21-71, 269-82-75, e-mail: anna-danekina@yandex.ru.

Optimization of size and creation time of parallel programs checkpoints

A.Yu. Polyakov, Danekina A.A.

An approaches to parallel programs incremental and differential checkpointing are stated. Hash based sequential and parallel algorithms are proposed and investigated. An efficiency of proposed approach for iterative and direct methods for solving complex tasks are shown. Good scalability of parallel version of proposed algorithm is shown.

Keywords: fault tolerance, checkpoint and recovery, incremental checkpoint, large scale systems.