Бахтин В.А., Жукова О.Ф.,
Крюков В.А., Сазанов Ю.Л.

Сравнительная отладка OpenMP-программ

Рекомендуемая форма библиографической ссылки: Сравнительная отладка OpenMP-программ / В.А.Бахтин [и др.] // Препринты ИПМ им. М.В.Келдыша. 2009. № 76. 24 с. URL: http://library.keldysh.ru/preprint.asp?id=2009-76

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Российской Академии наук

В.А.Бахтин, О.Ф.Жукова, В.А.Крюков, Ю.Л.Сазанов

Сравнительная отладка OpenMP-программ

Москва - 2009

В.А.Бахтин, О.Ф.Жукова, В.А.Крюков, Ю.Л.Сазанов. Сравнительная отладка OpenMP-программ. Препринт Института прикладной математики им. М.В. Келдыша РАН, 2009, 24 страницы, библиография: 15 наименований.

Обсуждаются принципы и проблемы сравнительной отладки при автоматизированном распараллеливании программ. Описывается экспериментальная версия системы сравнительной отладки для OpenMP-программ и результаты испытания системы на пакете тестовых программ.

V.A.Bakhtin, O.F.Zhukova, V.A.Krukov, Y.L.Sazanov. Comparative debugging of OpenMP programs. Preprint, Inst. Appl. Mathem., Russian Academy of Sciences, 2009, 24 Pages, 15 References.

Attitude Principles and problems of comparative debugging are considered with special stress on semi-automatic parallelization. Experimental version of comparative debugging system for OpenMP-programms is described, as well as results of its checking on test programs.

Содержание

1. Введение

Параллельные вычисления в настоящее время представляются единственным способом дальнейшего повышения производительности вычислительных систем, т.к. повышение тактовой частоты процессора, по-видимому, уже достигло предела. Однако всем известна сложность параллельного программирования, обусловленная в первую очередь непривычностью необходимого для этого «параллельного» мышления и сложностью учета принципиальной *недетерминированности* параллельных программ, которая порождает новый для прикладных программистов класс ошибок, с которыми при последовательном программировании им встречаться не приходилось. Известна и сложность отладки параллельных программ из-за:

- *невоспроизводимости* проявления ошибок, вызванной недетерминированностью выполнения;
- необходимости отслеживать состояние нескольких (очень многих) параллельных процессов;
- влияния отладочных средств на процесс выполнения (меняется время выполнения операторов, возможны внутренние синхронизации).

В настоящее время сложность параллельного программирования усугубляется разнообразием подходов, архитектур, несовместимых моделей параллелизма. Поэтому стала очень актуальной проблема автоматизации распараллеливания.

Автоматическое распараллеливание функционально близко машинно-зависимой оптимизации. Но есть существенная разница. *Полностью* автоматический распараллеливатель (как и обычный оптимизатор) должен *гарантировать* правильность преобразованной программы. Поэтому он должен быть «консервативным» в том смысле, что он может применять только те преобразования программы и только тогда, когда они гарантированно не меняют ее семантики. Для обычного оптимизатора такая «консервативность» вполне допустима, потому что худшее, что может случиться – это, что программа ускорится, скажем, не в 3 раза, а только в 2,5 раза. Оптимизация последовательных программ вообще редко дает ускорение больше, чем в 3-4 раза. Но для параллельных программ речь идет о возможности ускорения в *сотни* раз и поэтому «упущенная выгода» из-за излишнего консерватизма распараллеливателя совершенно неприемлема.

Неконсервативный распараллеливатель может принимать и «сомнительные» решения. Но, во-первых, он должен сообщать о них программисту. И, во-вторых, программисту должны быть предоставлены средства влиять на эти решения. Подобно тому, как у оптимизатора есть возможность включения/отключения некоторых видов оптимизации в виде опций командной строки или прагм компилятора (т.е. указаний в тексте программы). Заметим, что некоторые языки параллельного программирования (HPF [1], DVM [2], OpenMP [3]) построены как набор синтаксически оформленных указаний, вставляемых в текст последовательной программы. Сам компилятор может при этом считаться

«консервативным с точностью до ошибок» в указаниях программиста. Предельный случай управления распараллеливанием – это, конечно, ручное распараллеливание. Но в любом случае, если распараллеливатель не консервативен, т.е. не гарантирует правильность полученной программы, возникает необходимость ее *отладки*. Кроме того, в программе могут быть ошибки, не проявившиеся при ее последовательном выполнении, а проявляющиеся при параллельном выполнении.

Отладка программы, как правило, состоит в сравнении результатов ее работы (в т.ч. промежуточных) с некоторыми «эталонными». Автоматическая сравнительная отладка в качестве «эталонных» использует результаты работы некоторой «правильной» версии программы для отладки ее модифицированной версии. В случае *распараллеливателя* в качестве «правильной» («эталонной») версии программы можно взять ее исходный последовательный вариант, т.е. речь может идти об отладке *относительно* исходной программы (*relative* debugging). Такой способ отладки в некоторой мере допускает автоматизацию. Ниже описывается этот метод отладки – **сравнительная отладка –** и разрабатываемая система сравнительной отладки OpenMP-программ. Сравнительная отладка заключается в сравнении значений переменных в сопоставимые моменты выполнения двух версий программы (в данном случае *последовательной* и *параллельной*).

Метод сравнительной отладки впервые был предложен для последовательных программ в отладчике Guard [4,5], а для параллельных программ в DVM-системе [6,7]. Главное отличие между ними заключается в том, что в отладчике Guard сам пользователь задает точки в программах, в которых надо сравнивать указанные данные, а в DVM-системе и точки и сравниваемые данные определялись автоматически.

Основные этапы автоматической сравнительной отладки это: накопление промежуточных результатов (трассировка) и сравнение их с эталонными, причем сравнение может выполняться как динамически, так и с заранее собранной «эталонной» трассой. Для трассировки и (динамического) сравнения программа модифицируется в двоичном или исходном виде. В описываемой системе отладки используется универсальная инструментация source-to-source.

Описываемая ниже система сравнительной отладки для **OpenMP-программ** продолжает линию и использует опыт разработки сравнительных отладчиков для DVM [7] и MPI-программ [8].

2. Опыт и проблемы использования сравнительной отладки в DVM-системе

Первоначально сравнительная отладка в DVM-системе мыслилась как *полная* трассировка всех обращений к переменным и сравнение трасс до «*первого* расхождения». Для получения и сравнения или проверки трасс программа *инструментируется* – дополняется вызовами подпрограмм *трассировщика*. Инструментируются обращения к данным для записи читаемых и/или записываемых значений переменных и последовательные и параллельные программные конструкции для идентификации «момента» выполнения присваивания.

Эта простая идея показала свою эффективность на простых модельных задачах, но наткнулась на два препятствия: **ресурсы** и **точность**. Рассмотрим подробней суть этих проблем и как они решались в системе DVM.

2.1. Ресурсы

После полной инструментации каждый оператор окружается несколькими вызовами подпрограмм трассировщика, которые должны сформировать записи с читаемыми и записываемыми значениями переменных. Неудивительно, что при этом программа значительно замедляется. В [9,10] приводятся данные экспериментов на программах пакета NAS. Замедление уже только от инструментации составляет 50-100 раз. Объемы трассы – несколько десятков байтов на каждый выполненный оператор присваивания – тоже оказались совершенно неприемлемыми для реальных программ (средний размер трасс > 6.57e+12 байт, т.е. терабайты, и при среднем времени выполнения исходных программ ~15 сек. – среднее время сбора трасы ~28000 сек., т.е. увеличивается в 2000 раз!). Поэтому полная сравнительная отладка оказалась применимой только на «модельных» данных, которые не всегда доступны.

2.2. Точность, или “допустимые” отличия

Другая проблема, с которой столкнулась система отладки, – «допустимое» несовпадение значений переменных. Такими переменными являются, например, *редукционные переменные*. При вычислении суммы элементов распределенного массива меняется порядок операций: сначала вычисляются локальные суммы, а потом они суммируются в непредсказуемом порядке. Результаты при этом получаются разными (типично расхождение в 1-2 младших разрядах, хотя в принципе могут различаться сколь угодно сильно), но, с точки зрения программиста, эти результаты могут быть одинаково допустимыми. Редукционные операции создают четыре проблемы для сравнительной отладки («*ложные тревоги*»):

1. значения редукционной переменной на промежуточных итерациях не совпадают, потому что вычисляются только частичные суммы или максимум ищется в другом диапазоне и т.п.;
2. окончательное значение суммы, как сказано выше, может отличаться (недетерминизм *в слабом смысле*);
3. отличие в одной переменной может сразу распространиться на многие другие (например, если посчитана норма вектора и затем вектор нормируется);
4. а если это значение используется в критерии окончания итераций или при выборе ветви вычислений, то дальше могут быть выбраны разные ветви и трассы вообще могут оказаться несопоставимыми (недетерминизм *в сильном смысле*).

2.3. Методы преодоления трудностей

Для сокращения *времени работы и объема трассы* в DVM-системе были первоначально предложены и реализованы средства управления трассировкой:

- выборочная трассировка, например: только запись, только распределенные массивы, только указанные секции массивов и т.п.;
- локализация инструментации, т.е. выделение неинструментируемых фрагментов программы;
- предварительная оценка объема трассы, получение так называемого «файла циклов» и затем его ручная коррекция для отключения трассировки на определенных итерациях и т.д.

Эти методы оказались трудоемкими в использовании и недостаточно эффективными. Поэтому в DVM-системе были предложены и частично реализованы новые, дополнительные средства:

- автоматический выбор итераций для трассировки: «границы» и «уголки»;
- генерация двух тел цикла: одно без вызовов трассировщика, другое инструментировано; при этом на выбранных для трассировки итерациях работает инструментированное тело цикла, на остальных – исходная программа;
- условный вызов подпрограмм трассировщика (т.е. перед вызовом подпрограммы трассировщика проверяется необходимость ее вызова);
- трассировка «контрольных сумм» массивов по завершении цикла вместо трассировки элементов массивов при выполнении тела цикла.

Полученные результаты отражены в [9, 10] и оценены как достаточно успешные (с точки зрения эффективности), причем генерация двух тел циклов не дала существенного выигрыша по сравнению с условным обходом вызовов трассировщика.

Проблема *редукционных переменных* решалась так:

- редукционные переменные распознаются (т.к. они описаны в директивах DVM), и либо обращения к ним в теле цикла не трассируется, либо уже в трассе соответствующие записи не сравниваются;

- трассировка и/или сравнение значений выполняется с некоторой заданной *точностью*. Ошибка фиксируется, только когда различие становится достаточно велико. Точность сравнения может задаваться программистом;
- в режиме сравнения реального выполнения одного варианта программы с трассой выполнения другого варианта реально вычисленное значение редукционной переменной *заменяется* ее значением из «*эталонной*» трассы.

Эти приемы позволили подавить «ложные тревоги», связанные с редукционными переменными.

Но опыт показал, редукционные переменные являются не единственной причиной «ложных тревог».

3. Инструментация программ

Для отладочных средств, ориентированных на OpenMP, был разработан универсальный интерфейс инструментации [11]. *Универсальность* его состоит в возможности подключения к единожды инструментированной программе разных библиотек, подсистем. Например: динамического контроля, накопления трассы, сравнения (проверки) трассы. Основное его отличие от предыдущих версий инструментации – разделение статической и динамической информации. Вся необходимая статическая информация об операторах, конструкциях, директивах, данных записывается в виде символьных *контекстных строк* [12]. Контекстная строка содержит:

- тип контекстной строки;
- имя файла, номер строки в файле, а для конструкций номер первой и последней строки;
- идентификатор переменной, тип переменной, ранг массива;
- информацию из клауз OpenMP-директив.

Инструментатор формирует файл `DBG_INIT.H`, в котором содержатся все контекстные строки в виде аргументов вызовов подпрограммы регистрации контекстной строки. При инициализации трассировщика все контекстные строки регистрируются и после этого в вызовах отладчика используются их дескрипторы (номера). Все контекстные строки выводятся в начале файлов таблицы итераций и трассы. Визуализатор различий показывает контекстные строки и использует их для доступа к исходному тексту.

Пример контекстных строк (из трассы программы `JAC.fdv`). Первое число в каждой строке – количество символов в основной части. Поля разделяются символом ‘*’. Поля имеют вид: <*имя поля*> = <*значение*>.

```
     ....
     93*type=arr_name*file=jac.fdv*line1=3*name1=a*vtype=3*rank=
2*isindata=0*isincommon=0*isinsave=0*
     88*type=var_name*file=jac.fdv*line1=3*name1=eps*vtype=3*isi
ndata=0*isincommon=0*isinsave=0*
     ....
     другие описания данных
```

```
      ....
      37*type=file_name*file=jac.fdv*line1=33*
      45*type=parallel*file=jac.fdv*line1=36*line2=49*
      44*type=omploop*file=jac.fdv*line1=37*line2=48*
      44*type=seqloop*file=jac.fdv*line1=41*line2=48*
      37*type=file_name*file=jac.fdv*line1=42*
      37*type=file_name*file=jac.fdv*line1=44*
      37*type=file_name*file=jac.fdv*line1=46*
      44*type=seqloop*file=jac.fdv*line1=50*line2=73*
      37*type=file_name*file=jac.fdv*line1=51*
      96*type=parallel*file=jac.fdv*line1=52*default=shared*priva
te=i,j*redop=max*reduction=eps*line2=70*
      ....
      и т.д.
```

4. Оценка ресурсов для сравнительной отладки

На базе этого интерфейса инструментации сначала было проведено **исследование применимости** сравнительной отладки с использованием трассировки для реальных программ, при этом в какой-то мере подтверждены оценки сравнительного отладчика DVM. В качестве «реальных» программ использовался инструментированный пакет тестов NAS-OMP класса A [13]. Кроме них в тестовый пакет были включены тест JAC и программа рассчета солнечной конвекции CONVD [14]. Так как целью исследования была **оценка** времени и объема трассы, а **реальные** времена очень велики и объемы трасс превышают все логические и физические ограничения на размер файлов, использовался специально разработанный **макет** трассировщика. Кроме того, инструментированные программы **модифицировались** для изучения влияния вариантов инструментации.

4.1. Характеристики исходных программ пакета

В таблице приведены времена (в секундах) выполнения задач из пакета с оптимизацией по умолчанию (opt=2) и без оптимизации и их отношение (т.е. *качество оптимизации*) для двух вариантов задач: стандартного (class A) и сокращенного: уменьшены число итераций (до 2) или размер массива.

	class A opt=2	Class A Opt=0		Niter=2 Opt=2	Niter=2 opt=0	
JAC	0.1	0.1		0.0	0.0	
CG	15.4	28.2	~2	12.2	26.8	~2
CONVD	76.5	153.0	~2	96.7	160.2	~2
FT	51.9	236.5	~4.5	182.2	62.4	
MG	33.1	134.7	~4	33.7	68.2	~2
EP	264.9	365.3	~1.3	0.1	0.1	~
SP	337.4	1647.6	~5	6.7	13.3	~2
BT	494.1	2104.7	~4.5	0.2	0.2	~
LU	523.0	1864.7	~3.5	8.8	15.7	~2

4.2. Эффект инструментации

При инструментации размер (число строк) программы увеличивается в 4..10 раз. Число порождаемых контекстных строк сравнимо с числом строк исходного текста (на програмах пакета – от 40% до 110%). Замер времени выполнения производился на *макете трассировщика* с «пустыми» подпрограммами*:*

```
SUBROUTINE DBG_xxxxx ( ... )
RETURN
END
```

В таблице приводится время выполнения и его отношение ко времени выполнения соответствующего исходного, неинструментированного варианта программы (*замедление*).

	class A				Niter=2			
	opt=2		opt=0		opt=2		Opt=0	
JAC	61.1		79.8		1.1		1.4	
CG	111.4	~7	123.0	~4.5	62.3		63.5	~2.5
CONVD	357.1	~5	399.0	~3.5	463.3	(?)	517.3	(?)
FT	509.2	~10	680.7	~3	446.5		543.2	?
MG	331.7	~10	492.9	~3.5	178.8	~5	227.0	~3
EP	622.2	~2.5	650.1	~2	0.1		0.1	
SP	5150.7	~15	6126.1	~4	51.8	~7	56.1	~4
BT	9157.9	~18	12155.2	~6	1.2		1.4	
LU	7102.7	~13	9409.4	~5	88.9	~10	118.7	~8

Из таблицы видно, что инструментация практически блокирует оптимизацию компилятора: эффект оптимизации – 10-20% (для исходных – 3-5 раз). По сравнению с оптимизированной исходной программой вызовы пустых подпрограмм макета трассировщика замедляют программу на порядок.

4.3. Оценка требуемых ресурсов сверху

Оценка сверху для полного объема трассы получена *подсчетом числа обращений* к трассировщику. Прогонялись те же варианты программ. Ниже приводятся результаты для (class=A, opt=2). Подпрограммы макета трассировщика заменены на следуюшие:

```
SUBROUTINE DBG_xxxxx ( ... )
tr_size = tr_size + 1
RETURN
END
```

Приведен приблизительный коэффициент замедления по сравнению с последовательной программой, оттранслированной с опциями по умолчанию. В таблице приведено также общее число вызовов трассировщика.

	Время выполнения	Замедление	Число вызовов
JAC	394.7		3 398 008 485
CG	783.6	~50	6 853 454 472
CONVD	1898.7	~35	16 252 464 903
FT	3389.8	~60	27 676 379 446
MG	1896.0	~60	16 164 913 753
EP	3429.4	~13	30 662 223 057
SP	30697.8 ! ~ 8 часов	~90	263 717 040 013
BT	50531.2 ! ~ 14 часов	~100	421 049 218 080
LU	(выполнение прервано после 10 часов работы)		

Как оказалось, подпрограммы трассировщика с *одним* выполняемым оператором замедляют еще в 5-10 раз. Подпрограммы реального трассировщика вряд ли будут иметь меньше 100 операторов. Это может добавить еще 1-2 порядка. Этот результат по порядку подтверждает приведенную выше оценку для трассировщика DVM (28000 сек. против 15 сек.)

Оценка объема трассы тоже довольно близка: из расчета 10-20 байтов на запись (на вызов) дает десятки гигабайт, а для SP и BT уже терабайты.

4.4. Оценка доли трассировки данных

Для этого набора измерений инструментированные программы были модифицированы. Из них были **удалены** вызовы для трассировки чтения-записи переменных. Кроме того были разделены общий счетчик вызовов и счетчик вызовов для трассировки итераций (DBG_OMP_LOOP_EVENT и DBG_LOOP_EVENT).

Подробные результаты здесь не приводятся. А общие выводы таковы:

- трассировка данных составляет до 95% общего времени;
- доля трассировки итераций циклов достигает 99% и даже 99,8%, что и неудивительно. Поэтому в реальном трассировщике были разработаны специальные приемы для сокращения этого времени;
- однако для SP, BT и LU оно составило только 60-80%, потому что самые внутренние циклы оказались очень короткими (меньше 5 итераций) и становится заметным вклад регистрации начала/окончания цикла.

4.5. Оценка покрытия (статистика использования контекстных строк).

В этой серии измерений подсчитывалось (в массиве iOTCSTAT) и по завершении программы выводилось число использований каждой (операторной) контекстной строки. Тем самым определяется «операторное покрытие программы» и статистика выполнения операторов.

С точки зрения инструментации (и времени выполнения) это некоторый промежуточный вариант между двумя предыдущими, а именно, из программы удалены не все вызовы трассировки обращений к переменным, а только трассировка чтения.

4.6. Оценка требуемых ресурсов снизу

И, наконец, была получена оценка «снизу», т.е. оценка требуемых ресурсов при максимальном сокращении трассировки, сопоставимом с тем, что было предложено для сравнительной отладки DVM. Для этого инструментированные программы были преобразованы так, чтобы они могли использовать счетчики *использования* контекстных строк (т.е. выполнения вызовов) для «отключения» вызовов трассировки после определенного числа вызовов (в данном случае – после 1000 вызовов) следующим образом:

```
if( iOTCSTAT(33,0) < 1000 ) then
  call dbg_write_end(istat_mp(33),ithreadid,a(i,j),istat_mp(1))
endif
```

Результат таков:

- время выполнения без оптимизации практически совпадает со временем выполнения неоптимизированной исходной программы;
- время выполнения с opt=2 примерно в два раза больше, чем выполнение оптимизированной исходной программы – здесь нет противоречия с указанным выше замедлением в 10-50 раз: в этих тестах удалены вызовы для трассировки чтения переменных, это сокращает число вызовов в несколько раз;
- объем трассы оценивается уже только десятками мегабайт.

5. Описание системы сравнительной отладки

В описываемой системе сравнительной отладки для OpenMP-программ применяется выборочная трассировка при помощи динамического включения и выключения трассировки по предварительно сформированной таблице трассируемых итераций. Этот дополнительный этап обработки программы (построение таблицы итераций) предшествует этапам трассировки и сравнения. Выборочная трассировка сокращает объем трассы.

Для достижения приемлемой эффективности инструментированная программа преобразуется специальным образом («реинструментируется») как для построения таблицы итераций, так и для накопления и сравнения трассы.

Таким образом, сравнительная отладка состоит из этапов:

1. инструментация (см. выше);
2. *построение таблицы итераций*:
   - реинструментация программы;
   - компиляция программы;
   - выполнение (на одной нити или без OpenMP);
3. *сбор трассы*:
   - реинструментация программы;
   - компиляция (в режиме OpenMP);
   - выполнение на двух (или более) нитях;
4. *упорядочение трассы*, полученной при параллельном выполнении;
5. *проверка трассы* (возможны варианты):
   - выполнение той же программы на одной нити (или без OpenMP) в режиме проверки трассы;
   - сбор трассы на одной нити (или без OpenMP) и сравнение трасс стандартными средствами системы (fc или diff);
6. просмотр найденных отличий интерактивным *визуализатором различий*.

5.1. Построение таблицы итераций

Для уменьшения времени сбора трассы и ее размера используется выборочная трассировка. Для этого перед шагом сбора трассы сначала строится таблица итераций, которые должны трассироваться. В описываемой версии системы трассировка выполняется:

- на «уголковых» итерациях, т.е. итерациях, на которых переменные всех циклов принимают крайние значения – первое и последнее из своих диапазонов;
- на итерациях, на которых хоть один оператор выполняется в первый раз (для обеспечения «операторного покрытия» программы).

Реинструментация. При построении таблицы итераций не используются вызовы для регистрации данных. Поэтому они удаляются из текста инструмен-

тированной программы. Для регистрации первого выполнения оператора присваивания достаточно вызова DBG_WRITE_END. Поэтому вызовы по чтению переменных тоже удаляются. При первом выполнении оператора (вызове трассировщика с контекстной строкой, соответствующей этому оператору) этот факт отмечается в таблицах трассировщика, доступных из отлаживаемой программы. Вызов трассировщика сделан условным с предварительной проверкой этого признака, т.е. для каждого оператора присваивания трассировщик за все время выполнения программы вызывается только один раз.

Как показали предварительные замеры, вызовы подпрограммы DBG_LOOP_EVENT составляют весьма значительную долю накладных расходов. Заметим, что номер итерации нужен только в момент регистрации первого выполнения. Эти вызовы из программы тоже удаляются. Вместо них вставляется запись текущего значения переменной цикла в определенную глобальную переменную трассировщика. Это значение будет использовано, если понадобится регистрация первого выполнения некоторого оператора в теле цикла. Подпрограммы DBG_BEFORE_LOOP, DBG_AFTER_LOOP (и аналогичные для OMP DO) управляют стеком текущих значений переменных циклов.

Для построения таблицы итераций предварительно реинструментированная (**re_dbg.exe –1 ...**) программа должна быть скомпилирована в режиме OpenMP вместе с модулем трассирощика **otcrts1.f90**.

Таблица итераций записывается в *формате трассы*, но с сокращенным множеством типов записей, т.е. представляет из себя сокращенный протокол будущего параллельного или последовательного выполнения программы.

5.2. Накопление параллельной трассы

Для накопления параллельной трассы предварительно реинструментированная (**re_dbg.exe –2 ...**) программа должна быть скомпилирована в режиме OpenMP вместе с модулем трассировщика **otcrts2.f90**. Реинструментация для сбора и проверки трассы сводится к удалению некоторых вызовов и вставке условных операторов для динамического управления трассировкой.

При накоплении трассы используется сформированная таблица итераций. Таблица считывается и во время выполнения в компактном виде хранится в памяти. Каждая нить (независимо) отслеживает свое текущее положение в таблице итераций, входы и выходы из конструкций OpenMP и последовательных циклов, и при этом меняет признак текущего состояния трассировки («включена/выключена») для данной нити в зависимости от текущей выполняемой итерации и номеров итераций, записанных в таблице. Все вызовы для трассировки данных сделаны условными с проверкой этого признака. В последовательном цикле все итерации выполняет та же нить, в которой выполнен вызов DBG_BEFORE_LOOP, поэтому список трассируемых итераций она проходит последовательно. После каждой оттрассированной итерации известен номер следующей, на которой надо включить трассировку. Вызов подпрограммы регистрации номера итерации (DBG_LOOP_EVENT) сделан условным со сравнением текущего номера итерации со следующим указанным в таблице итераций.

Когда эта подпрограмма вызывается, она включает признак трассировки на текущую итерацию. Для параллельного цикла алгоритм несколько сложней: т.к. нить не может «пропустить» итерации до следующей трассируемой, потому что не известно для какой нити какая трассируемая итерация окажется следующей. Поэтому для параллельного цикла подпрограмма DBG_OMP_LOOP_EVENT должна вызываться на каждой итерации и должна проверять, нет ли в таблице итерации с текущим номером. Накладные расходы на просмотр таблицы могут стать значительными, если этот параллельный цикл является самым внутренним и коротким.

Записи трассы предварительно накапливаются в памяти в буферах (у каждой нити свой буфер) и, когда некоторый буфер заполнен, он выталкивается в файл. Только в этот момент, для доступа к одному и тому же файлу требуется синхронизация нитей. Максимальное число буферов и их размер можно задать при *компиляции* модуля трассировщика вместе с программой.

5.3. Упорядочение параллельной трассы

При запуске в параллельном варианте (с OpenMP) трасса содержит вперемешку блоки записей от разных нитей. Перед проверкой трассы она «упорядочивается», т.е. записи переставляются в таком порядке, в котором они получились бы при последовательном выполнении, а именно итерации параллельных циклов (**omp do**) выбираются из блоков записей, выданных разными нитями, в порядке изменения переменной цикла, а секции (конструкция **sections**), выполненные разными нитями переставляются в текстуальном порядке.

Отдельную проблему представляют операторы параллельного региона вне конструкций распределения работы: при параллельном выполнении они выполняются всеми нитями, а при последовательном – только одной. В принципе программа может быть написана так, что на этих участках в разных нитях будут выполняться совершенно разные участки кода (например, в зависимости от номера нити) и/или переменные (приватные) будут принимать разные значения. Это делает последовательную и параллельную трассы *несопоставимыми*. В текущей версии *предполагается* (и разрабатываемый автоматический распараллеливатель это должен обеспечивать), что на таких общих участках выполнения нитей выполняются одни и те же операторы и переменные принимают одни и те же значения. Программа упорядочения трассы сравнивает значения переменных. При несовпадении формируются сообщения в формате файла различий трасс (т.е. результата проверки, см. следующий раздел), а в качестве значения в упорядоченной трассе берется значение из нити **master**.

5.4. Проверка трассы

Проверку трассы выполняет та же программа, которая использовалась при ее получении, когда она обнаруживает в текущей директории файл трассы (со стандартным именем). Программа должна быть запущена на одной нити (или перекомпилирована без OpenMP).

При выполнении программы в режиме проверки трассы формируются такие же записи, как и при накоплении трассы, но вместо сохранения их в буфере и записи в файл, с ними последовательно сравниваются записи из проверяемой упорядоченной трассы. Включение/выключение проверки выполняется не по таблице итераций, а по проверяемой трассе. Так как трасса была упорядочена и OpenMP-циклы при проверке выполняются последовательно (на одной нити), то, как для последовательных, так и для параллельных циклов всегда известна следующая итерация, на которой надо включить сравнение, т.е. вызывать программы отладчика. На остальных итерациях эти вызовы обходятся.

При несовпадении записи из трассы с текущей записью в файл отличий (**OTC_DIFFS**) дописывается строка, содержащая для экономии только минимально необходимую информацию: номер записи в проверяемой трассе и «правильную» запись.

5.5. Визуализатор различий

При запуске визуализатор пытается прочитать файлы (в текущей папке): файл отличий OTC_DIFFS, трассу OTC_TRACE, файлы исходных текстов, на которые есть ссылки в контекстных строках. При успешном чтении файлов визуализатор сообщает их размер (в байтах и строках), а также количество контекстных строк. Затем, после нажатия любой клавиши, открывается основное окно, показанное на рисунке (использованы результаты прогона программы CG).

Окно визуализатора (алфавитно-цифровое 80x50) разбито на 4 горизонтальные панели: отличия (OTC_DIFFS), трасса (OTC_TRACE), контекстные строки (из начала OTC_TRACE), исходные тексты. Каждая панель содержит номер строки и данные. Текущая строка в каждой панели подсвечена. Активная панель также выделена.

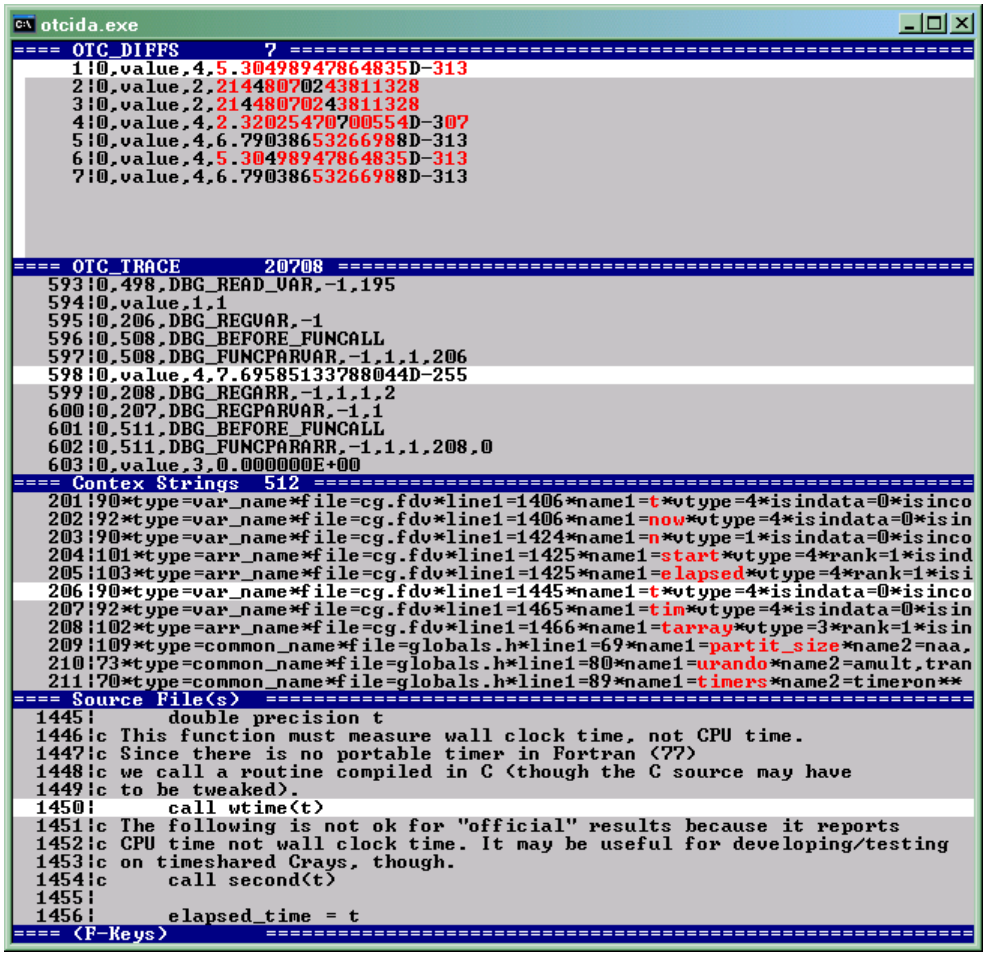
Первая (верхняя) панель содержит «эталонную» запись, с которой не совпала запись трассы. Для наглядности отличающиеся символы записей выделяются красным цветом. Номер неправильной строки в трассе, который присутствует в файле отличий, не отображается, но используется для позиционирования трассы во второй панели.

Вторая панель содержит соответствующий участок упорядоченной трассы. Несовпавшая запись является текущей. В примере на рисунке она установлена на запись с номером 598 (из 20708, как указано в заголовке). Несовпадающие символы также подкрашены.

Третья панель устанавливается на контекстную строку переменной, для которой обнаружено расхождение. На нее указывает последнее поле в записи трассы 597. Из этой записи (597) видно также, что выполнялся вызов трассировщика DBG_FUNCPARVAR – трассировка значения фактического параметра. Идентификатор переменной (“t”) выделен красным цветом.

Четвертая панель устанавливается на строку исходного текста. Для этого используется номер контекстной строки оператора (508) и параметры file и line1 в ней.

В данном случае обнаружено различие в значении переменной t, исполь-



зованной как параметр вызова функции wtime. Это выходной параметр и начальное значение его безразлично; переменная передается в функцию именно **для** инициализации.

Управление просмотром:

- прокрутка данных в активной панели: UP, DOWN, PGUP, PGDN, HOME, END:
- смена активной панели: LEFT, RIGHT;
- удаление (неинтересных, проанализированных) отличий: DEL;
- завершение работы: F10.

При прокрутке данных в текущей активной панели содержимое других панелей автоматически синхронизируется с ней (в порядке сверху вниз, т.е. отличия – трасса – контекстная строка – исходный текст).

6. Экспериментальная версия системы

Экспериментальная версия системы сравнительной отладки реализована в системе Windows и поставляется в виде директории, содержащей:

- системную поддиректорию OTC_SYS, которая содержит все необходимые exe-файлы и f90-файлы;
- поддиректории с тестами (JAC, BT, CG, EP, MG и т.д.), содержащие исходные тексты и результаты работы (в экспериментальной версии сохраняются все промежуточные файлы и выдачи программ);
- и головные bat-файлы для запуска одного или всех тестов, визуализатора и настройки окружения в корневой директории.

В экспериментальной версии системы полный запуск теста содержит все описанные выше шаги и кроме того предварительно выполняются (для замера времени) еще два шага:

- исходная программа компилируется в режиме OpenMP и выполняется на двух нитях;

- исходная программа компилируется как последовательная, выполняется сбор трассы на одной нити и сравнение трасс как текстовых файлов (команда fc).

Скрипты экспериментальной версии поддерживают обработку только одномодульных программ, имя исходного файла теста должно совпадать с именем поддиректории и он должен иметь расширение `fdv` (только один файл – <*тест*>\<*тест*>.fdv – инструментируется и компилируется).

Демонстрационная версия может использоваться для просмотра результатов, перезапуска тестов, перекомпиляции тестов и экспериментов со своей собственной (одномодульной!) программой. Для установки демонстрационной версии достаточно распаковать архив и затем:

- если архив содержит результаты пропуска теста, то можно их просмотреть, войдя в соответствующую поддиректорию и выполнив команду "`..\ida`";
- если нужно перезапустить тест (готовые exe-файлы из архива), то в корневой директории надо выполнить команды "`go_test` *`sub-dir`*" или "`go_all`";
- если нужно (изменить) и перекомпилировать тест, то нужно сначала исправить в "`setenv.bat`" путь к "`ifortvars.bat`" (или его аналогу) и затем в корневой директории выполнить команды "`runtest` *`sub-dir`*" или "`runall`";
- или создать собственный тест *`XXXX`*`.fdv`, записать его в директорию *`XXXX`* и выполнить команду "`go_test` *`xxxx`*", а при нормальном завершении теста команду выдать команду "`..\ida`".

7. Испытание системы на тестовых программах

В качестве тестовых программ использовались: JAC, семь программ из пакета NAS-OMP (BT, CG, EP, MG, LU, SP, FT, LU), вариант теста LU после текстовой подстановки процедур, программы CONVD и KINXYZ[15], а также несколько программ, изготовленных автоматическим распараллеливателем.

Продемонстрировано вполне приемлемое время выполнения и размеры трасс. Замедление при построении таблицы итераций – от 2 до 10 раз, при сборе и проверке трассы – не более 2 раз. Размер трассы – от 0,5 до 20 Мб. Найденных отличий – от 20 до 20000.

Сравнение трасс находит *на «правильных» программах* следующие (ожидаемые) отличия.

1. Тесты NAS в «отчете» выводят «Total threads», число нитей. Оно, естественно, отличается, и это отличие обнаруживается системой.
2. Инструментатором предусмотрена трассировка *всех* фактических аргументов перед вызовом программы. Но для рабочих массивов (это обычный стиль программирования на фортране-77) и выходных аргументов исходное содержимое безразлично, может различаться и фактически различается. Различие фиксируется системой.
3. Редукция (сумма). Система фиксирует значительное различие при вычислении частичных сумм и различие в младших разрядах после редукции.
4. В нескольких тестах NAS редукция по массиву выполняется “вручную” (в программе редукция никак синтаксически не отражена), а именно:
   - для внешнего общего массива создаются приватные копии;
   - в параллельном регионе сначала всеми нитями выполняется последовательный (неразделяемый) цикл инициализации своей приватной копии;
   - затем в параллельном (разделяемом) цикле в локальных массивах накапливаются частичные суммы и наконец
   - всеми нитями выполняется последовательный цикл добавления частичных сумм в общий разделяемый массив (во избежание коллизий, добавление, естественно, выполняется в контрольной секции).

   Система фиксирует различия при вычислении нитями частичных сумм в разделяемом цикле. На шаге *упорядочения трассы* фиксируется различие в последнем (неразделяемом) цикле при выполнении “того же” оператора в разных нитях. Затем при проверке трассы на этих же операторах фиксируется несовпадение с последовательной программой.
5. В тесте LU-OMP применяется синхронизация между нитями при помощи массива общих переменных (активное ожидание). На шаге упорядочения трассы фиксируется расхождение в нитях по операторам: некоторые нити выполняют цикл ожидания и (после выхода из цикла) присваивание синхронизационной переменной, а некоторые нити сразу продолжают вы-

полнение. В текущей версии системы это расхождение по выполняемым операторам влечет отказ от дальнейшего сравнения, но трасса до точки расхождения проверяется.

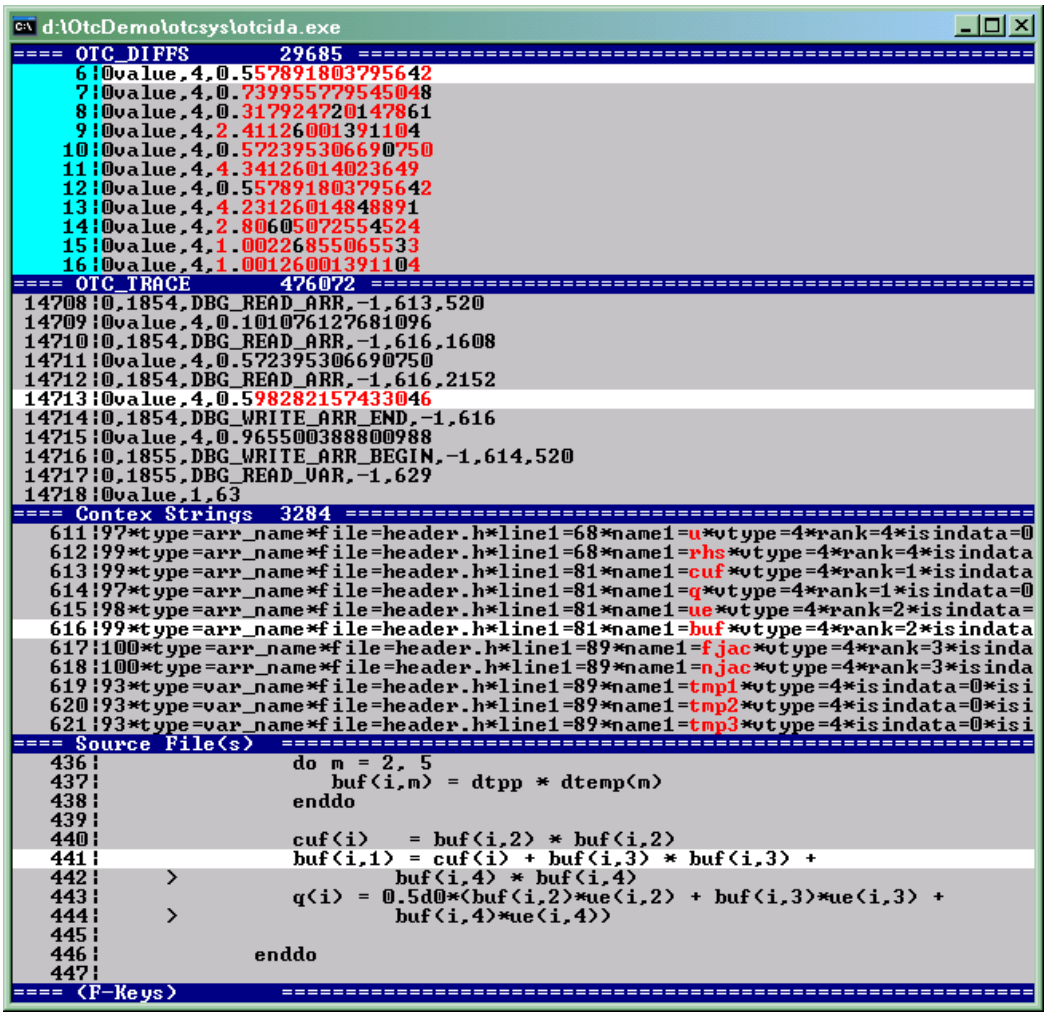
Поскольку все отличия однотипны, визуализатор позволяет достаточно быстро их проанализировать.

Для проверки способности системы находить реальные ошибки в тест BT была внесена ошибка, имитирующая реально сделанную автоматическим распараллеливателем: несколько переменных не были объявлены приватными.

```
...
CDVM$ ON forcing(i__45,j__44,k__43,*)
!$OMP PARALLEL PRIVATE (k__43,j__44,i__45,dble_333_34,zeta__47, dble_333
!$OMP*_35,eta__48,ii,dble_333_36,xi__49,m__29,m__46,dtpp)
!!!!$OMP*dtemp,cuf,buf,ue,q) FIRSTPRIVATE(jj)
!$OMP DO  SCHEDULE (STATIC)
...
```

Различий по сравнению с правильной программой найдено гораздо больше (29000 вместо 100). Несколько первых различий, как и в правильной программе, связаны с трассировкой значения входного параметра. Но уже для 6-го различия визуализатор показывает (см. рисунок), что в операторе 441 (4-я панель) при чтении (строка 14713: DBG_READ_ARR) из массива **buf** (3-я панель, строка 616) значения не совпали (в трассе 0.5982..., а должно быть 0.55789...). А в панели исходного текста видно, что в массив **buf** было присваивание в строках 436-438. Остальное должен додумать программист.

Если строку 9 в первой панели (различия) сделать текущей, то мы увидим, что различие возникло при чтении массива **ue**, который, как видно на панели исходного текста, используется в операторе 443. Присваивание было в строках 430-432; это можно увидеть, если перейти в панель исходного текста и полистать его.



8. Заключение

В данной работе обсуждаются сравнительная отладка параллельных программ, возникающие при ее применении проблемы и возможные способы их решения.

Описывается экспериментальная версия системы сравнительной отладки для OpenMP-программ, реализующая некоторые из этих решений. Система сравнительной отладки является частью комплекса программ для автоматизированного распараллеливания и отладки. Система базируется на универсальном инструментаторе OpenMP-программ и состоит из программ реинструментации, модулей построения таблицы представительных итераций, накопления и проверки трассы, программы упорядочения параллельной трассы и визуализатора различий.

Экспериментальная версия была испытана на пакете тестов NAS-OMP и нескольких реальных программах, и показала применимость для программ такого типа и размера. В процессе испытаний были выявлены некоторые проблемы, которые предполагается устранить в следующих версиях системы.

9. Литература

1. High Performance Fortran language specification // High Performance Fortran Forum. Scientific Programming. - 1993. - Vol. 2. - P.1-170.
2. Konovalov N. A., Krukov V. A., Mihailov S. N. and Pogrebtsov A. A. Fortran DVM - a Language for Portable Parallel Programs Development // Proceedings of Software For Multiprocessors and Supercomputers: Theory, Practice, Experience (SMS-TPE 94), Inst. for System Programming RAS. - Moscow, Sept. 1994. - P.124-133.
3. OpenMP Forum, “OpenMP: A Proposed Industry Standard API for Shared Memory Programming,” October, 1997. http://www.openmp.org.
4. Guard Parallel Relative Debugger. http://sourceforge.net/projects/guardsoft/
5. Abramson D.A., Sosic R. Relative Debugging using Multiple Program Versions // Intensional Programming I. Sydney: World Scientific. 1995.
6. http://www.keldysh.ru/dvm
7. В.А. Крюков, Р.В.Удовиченко, "Отладка DVM-программ", Препринт ИПМ им. М.В.Келдыша РАН №56, 1999
8. В.Ф. Алексахин, К.Н. Ефимкин, В.Н. Ильяков, В.А. Крюков, М.И. Кулешова, Ю.Л. Сазанов. Средства отладки MPI-программ в DVM-системе. Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», г. Новороссийск, 19-24 сентября 2005 г., стр 113-115
9. Крюков В.А., Кудрявцев М.В. Автоматизация отладки параллельных программ // Вычислительные методы и программирование. 2006. Том 7, раздел 2. 102-109.
10. М.В.Кудрявцев. “Автоматизация отладки параллельных программ”, диссертация на соискание ученой степени кандидата физ.-мат. наук, 2006 г.
11. В.А.Алексахин, В.О.Баринова, В.А.Бахтин, В.Д.Емельянов, В.А.Крюков, Ю.Л.Сазанов. Средства отладки OpenMP программ. // Труды Всероссийской научной конференции “Научный сервис в сети Интернет: решение больших задач”, сентябрь 2008 г., г. Новороссийск. – М.: Изд-во МГУ, 2008, стр. 281-285
12. B. Mohr, A. Malony, S. Shende, F.Wolf, “Design and Prototype of a Performance Tool Interface for OpenMP”, *The Journal of Supercomputing*, 23, 105–128, 2002
13. H.Jin, M.Frumkin and J.Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011. http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf
14. S.Ustyugov. Three Dimensional Numerical Simulations of Near Surface Solar Convection with Realistic Physics. Proceedings of the SOHO 14 / GONG 2004 Workshop (ESA SP-559). "Helio - and Asteroseismology: Towards a Golden Future". 12-16 July, 2004. New Haven, Connecticut, USA
15. A.Voronkov, V.Arzhanov. REACTOR - Program System for Neutron-Physical Calculations. Proc. International Topical Meeting: Advances in Mathematics, Computations, and Reactor Physics, USA, Vol5, April 28 - May 2, 1991