Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Образовательный комплекс «Технологии разработки параллельных программ»

Лабораторная работа: Отладка параллельной программы с использованием Intel Thread Checker

Разработчики: А.В.Сысоев, И.Б. Мееров

Нижний Новгород
2007

Содержание

Введение

Говорят «Любая программа содержит хотя бы одну ошибку». Применительно к параллельному программированию этот тезис можно переформулировать так «Параллельное программирование начинается с параллельных ошибок». Действительно, процесс создания параллельной программы, либо с нуля на основе параллельного алгоритма, либо путем распараллеливания существующей реализации, вносит в разработку программной системы дополнительные сложности, которые неизбежно приводят к появлению дополнительных по сравнению с программированием последовательным ошибок. Конечно, на основе некоторого опыта можно выделить типовые ошибки, встречающиеся в процессе разработки параллельной программы и, вооружившись этим знанием, пытаться в дальнейшем не допускать их, однако на практике… Логика параллельных программ существенно более сложна, чем у программ последовательных. Известно, что поиск ошибки начинается с обнаружения ее наличия. В параллельной программе, характер выполнения которой весьма часто носит «налет» случайности, ошибка может проявляться один раз на сотню и более запусков. Как «поймать» столь неуловимую вестницу беды в программе? Остро необходима инструментальная поддержка процесса отладки.

В настоящей лабораторной работе рассматривается один из инструментов отладки, основанный на сборе и автоматическом анализе информации по результатам выполнения программ и предназначенный для многопоточных и OpenMP программ – Intel® Thread Checker (ITC).

В качестве полигона для демонстрации ошибок и возможностей инструментов по их поиску и анализу используются: классическая задача скалярного умножения векторов; задача Дирихле для уравнения Пуассона, решаемая методом Гаусса-Зейделя; известная задача об обедающих философах и задача «о роботе».

1. Методические рекомендации

1.1. Цели и задачи работы

Целью данной лабораторной работы является приобретение практических навыков отладки параллельных программ для систем с распределенной памятью, использующих для организации параллелизма либо механизм потоков, либо технологию OpenMP.

Данная цель предполагает решение следующих задач:

- ознакомление с классификацией ошибок, возникающих при разработке многопоточных программ (см. документ «ITC – описание инструмента»);
- изучение функциональности инструмента отладки Intel Thread Checker (см. документ «ITC – описание инструмента»);
- изучение ряда учебных примеров, направленных на демонстрацию принципов отладки параллельных программ при помощи инструмента отладки Intel Thread Checker;
- самостоятельная разработка и отладка параллельных программ, использующих Windows Threads или технологию OpenMP.

1.2. Структура работы

Данный документ состоит из введения, двух разделов, списка дополнительных заданий и списка литературы. Во введении обосновывается актуальность инструментальной поддержки в процессе отладки параллельных программ. В первом разделе приводятся методические рекомендации к лабораторной работе: формулируются цели и задачи, системные требования, рекомендации по проведению занятий. Во втором разделе изучается отладка параллельных программ с демонстрацией характерных ошибок и изучением методов их обнаружения и устранения. Изучение проводится на специально подобранных примерах. Для обнаружения ошибок используется инструмент отладки Intel Thread Checker. В заключение приводятся задания для самостоятельной проработки, а также использованная и рекомендуемая литература.

1.3. Системные требования

В документации к ITC [2] приводятся следующие системные требования:

1.3.1. Аппаратное обеспечение

Минимальные требования

- процессор Pentium® 4;
- ОЗУ 512 Мб;
- свободное дисковое пространство 300 Мб.

При практическом использовании ITC рекомендуются повышенные требования к минимально-необходимым аппаратным ресурсам (для достижения приемлемых показателей оперативности работы):

- процессор Intel® Pentium® 4, поддерживающий технологию Hyper-Threading, или процессор Intel® Xeon®;
- ОЗУ 2 Гб.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

1.3.2. Программное обеспечение

- Microsoft Windows XP Professional, или Microsoft Windows Server 2003 или Microsoft Windows XP Professional x64 Edition[1].
- Intel® VTune™ Performance Analyzer версии 7.2 или выше.
- Microsoft Internet Explorer 6.0 или выше.
- Microsoft Visual C++ 6.0 или выше.
- Adobe® Reader®.

Для анализа OpenMP-программ и компиляторной инструментации требуется один из компиляторов:

- Intel® C++ Compiler 8.1 для Windows для архитектуры IA-32, Package ID: w_fc_pc_8.1.023 или выше.
- Intel® C++ Compiler 9.1 для Windows для архитектуры Intel® EM64T.
- Intel® Fortran Compiler для Windows 8.1, Package ID: w_fc_pc_8.1.023 или выше.

1.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется придерживаться следующей последовательности изучения материала:

- Рассмотреть классификацию типовых ошибок при разработке программ, использующих потоки, в том числе OpenMP-программ.
- Получить общее представление об инструменте отладки Intel Thread Checker (см. документ «Описание инструмента»).
- Последовательно изучить 4 описанных в данном документе учебных примера, демонстрирующих разные ошибочные ситуации и приемы их диагностики и устранения. При изучении примеров 2-4 по возможности рекомендуется предложить слушателям самостоятельно найти ошибки в приведенных программных реализациях, используя Intel Thread Checker, а затем исправить их.
- Перейти к самостоятельному выполнению слушателями дополнительных заданий.

---

[1] Существует версия ITC и для операционных систем семейства Linux, не имеющая, правда, графического интерфейса и работающая только из командной строки.

2. Отладка OpenMP и многопоточных программ на примерах

2.1. Задача о скалярном произведении векторов

2.1.1. Постановка задачи

Задача скалярного умножения векторов отличается простотой математической постановки и очень не сложной программной реализацией (естественно речь идет о последовательной неоптимизированной версии). Вместе с тем, эта задача является алгоритмическим элементом других более трудоемких операций (умножение матрицы на вектор, матричное умножение), да и сама служит хорошим примером, на котором можно демонстрировать большое количество эффектов, появляющихся как при распараллеливании, так и при оптимизации кода.

Итак, пусть имеются вектора *a* и *b* размерности *n* (состоящие из *n* элементов). Скалярным произведением векторов *a* и *b* называется число *c*, получаемое как:

$$c = (a,b) = \sum_{j=1}^{n} a_j b_j \ .$$

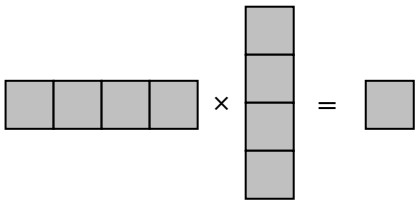


Рис. 1. Скалярное произведение двух векторов

Так, например, при умножении вектора $a = (1,2,3)$ на вектор $b = (3,2,1)$ получится, как нетрудно посчитать 10:

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \times \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} = 1*3 + 2*2 + 3*1 = 10$$

Рис. 2. Пример скалярного произведения векторов

2.1.2. Последовательная реализация

Алгоритм вычисления скалярного произведения полностью описывается его формулой, поэтому приведем сразу его код:

```
sum_all = 0;
for (i = 0; i < Size; i++)
{
  sum_all += x[i] * y[i];
}
```

Необходимые объявления переменных, а также ввод данных здесь рассматривать не будем (см. проект **Scalar**).

2.1.3. Параллельная реализация 1

Алгоритм скалярного умножения является примером с практически идеальным теоретическим распараллеливанием. Каждая операция умножения компонент векторов может быть выполнена независимо. Единственное «узкое» место – формирование итоговой суммы – может быть легко устранено накоплением результатов в локальных для каждого потока переменных (данных подход демонстрируется в параллельной реализации 2). Однако на практике получить при расчете скалярного произведения хорошие показатели ускорения весьма непросто в силу чрезвычайной легковесности

операции в распараллеливаемом цикле. Поскольку в данный момент нас интересует лишь создание работоспособной параллельной версии (с использованием ITC при необходимости), поэтому вопросы производительности мы затрагивать не будем.

Итак, мы имеем цикл с регулярной структурой и одинаковой вычислительной сложностью итераций. Распараллеливание его средствами OpenMP не представляет никакого труда.

```
sum_all = 0;
#pragma omp parallel for
for (i = 0; i < N; i++)
{
  sum_all += up[i] * vp[i];
}
```

Некоторый произвол возможен в том, как разделить вычисления между потоками. Первая схема – каждый поток обрабатывает непрерывный фрагмент умножаемых векторов. Вторая – вектора делятся между потоками поэлементно (или поблочно). Каждая схема может быть реализована с использованием параметра **`schedule`**. Провести соответствующие эксперименты предоставляем читателю самостоятельно.

В представленном выше коде мы сознательно допустили достаточно очевидную ошибку. Посмотрим, как с ней справится ITC.

2.1.4. Анализ параллельной реализации 1

Откройте проект **Scalar**, последовательно выполняя следующие шаги:

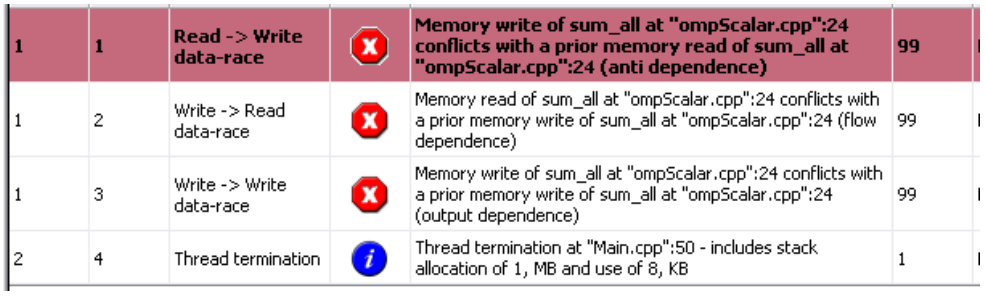
- запустите приложение **Microsoft Visual Studio 2005**,
- в меню **File** выполните команду **Open→Project/Solution…**,
- в диалоговом окне **Open Project** выберите папку **C:\ITCLabs\Scalar**,
- дважды щелкните на файле **Scalar.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **Scalar.cpp**. После этих действий программный код, с которым предстоит работать, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

Далее выполните действия, описанные в пункте 7.3 документа «ITC – Краткое описание.doc», чтобы подготовить программу для анализа в ITC.

И, наконец, запустите ITC, создайте в нем новый проект и укажите путь к исполняемому файлу **C:\ITCLabs\Scalar\Debug\Scalar.exe**.

Нажмите кнопку **Finish**. После этого запустится ITC, произведет инструментацию программы и начнет анализ. Результатом станет появление следующей диагностики:



1	**1**	**Read -> Write data-race**		**Memory write of sum_all at "ompScalar.cpp":24 conflicts with a prior memory read of sum_all at "ompScalar.cpp":24 (anti dependence)**	**99**	
1	2	Write -> Read data-race		Memory read of sum_all at "ompScalar.cpp":24 conflicts with a prior memory write of sum_all at "ompScalar.cpp":24 (flow dependence)	99	
1	3	Write -> Write data-race		Memory write of sum_all at "ompScalar.cpp":24 conflicts with a prior memory write of sum_all at "ompScalar.cpp":24 (output dependence)	99	
2	4	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1	

Рис. 3. Результат анализа параллельной реализации 1 скалярного умножения

Как и следовало ожидать источник неприятностей – переменная **`sum_all`**, на что недвусмысленно указывает ITC, сигнализируя о гонке данных при работе с этой переменной.

Для исправления представленного выше кода требуется лишь одна модификация – переменная, в которую накапливается сумма, должна быть сделана локальной для каждого потока. В противном случае во избежание конфликта доступа придется заключать операцию суммирования в критическую секцию, что сведет эффект от распараллеливания к нулю. Естественно по окончании работы потоков необходимо собрать из локальных сумм общий результат. Реализовать эту схему в OpenMP можно различными

способами, рассмотрим наиболее быстрый вариант, использующий параметр **`reduction`** директивы **`parallel for`**.

Вносим исправления:

```
sum_all = 0;
#pragma omp parallel for reduction(+:sum_all)
for (i = 0; i < N; i++)
{
  sum_all += up[i] * vp[i];
}
```

Повторно запускаем активность в ITC и убеждаемся в отсутствии ошибок:

Re...	ID	Short Description	Severity	Description	Count
1	1	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 4. Результат анализа параллельной реализации 1 после исправления

Отметим, что параметр **`reduction(+:sum_all)`** решает обе заявленные выше задачи (локализации переменной суммирования и подсчета итоговой суммы) и это наиболее короткий из возможных вариантов.

2.1.5. Параллельная реализация 2

В дополнение рассмотрим еще один несколько искусственный вариант скалярного умножения, который, тем не менее, позволит нам продемонстрировать некоторые типичные ошибки, нередко допускаемые при создании OpenMP программ.

Итак, пусть параллельная схема вычисления скалярного произведения выглядит следующим образом.

```
#pragma omp parallel private(rank, sum)
{
  rank = omp_get_thread_num();
  if (rank == 0)
  {
    NumThreads = omp_get_num_threads();
    x = CreateVector(N);
    y = CreateVector(N);
  }
  sum[rank] = 0;
  #pragma omp for
  for (i = 0; i < N; i++)
  {
    sum[rank] += x[i] * y[i];
  }
}
sum_all = 0;
for (i = 0; i < NumThreads; i++)
{
  sum_all += sum[i];
}

DeleteVector(x);
DeleteVector(y);
```

Нулевой поток «занимается» выделением памяти под данные и их инициализацией, после чего начинается собственно расчет. Представленная схема хоть и выглядит для задачи скалярного умножения чересчур усложненной, но в других ситуациях вполне может с успехом применяться. Отметим также, что здесь использовано другое решение для локализации суммы в потоках – массив **`sum`**.

Попробуйте самостоятельно найти в представленном коде ошибки, прежде чем мы перейдем к анализу.

2.1.6. Анализ параллельной реализации 2

Откройте проект **Scalar2**. Соберите программу в конфигурации **debug**. Прежде всего, посмотрим на результаты работы последовательной и параллельной версии. Зададим размер векторов равный 100 и запустим программу.

```
Sequential Time: 0.000117100
Sequential scalar product result is 328350.0

Parallel Time: 0.004261040
Parallel scalar product result is -92559631349317831000000000000000000000000000000000000000000000000000.0
```

Рис. 5. Результат запуска параллельной реализации 2

Итак, результаты запуска однозначно указывают на некорректность параллельной версии. Приступаем к поиску ошибок.

Снова выполните действия, описанные в пункте 7.3 документа «ITC – Краткое описание.doc», чтобы подготовить программу для анализа в ITC. Запустите ITC, создайте в нем новый проект и укажите путь к исполняемому файлу **C:\ITCLabs\Scalar2\Debug\Scalar2.exe**.

Нажмите кнопку **Finish**. После этого запустится ITC, произведет инструментацию программы и начнет анализ. Результатом станет появление следующей диагностики:

1	1	OpenMP - use of unsupported...		OpenMP - use of unsupported API at "ompScalar.cpp":60	1
1	2	Write -> Write data-race		Memory write of NumThreads at "ompScalar.cpp":63 conflicts with a prior memory write of NumThreads at "ompScalar.cpp":63 (output dependence)	1
1	3	Write -> Write data-race		Memory write of x at "ompScalar.cpp":64 conflicts with a prior memory write of x at "ompScalar.cpp":64 (output dependence)	1
1	4	Write -> Write data-race		Memory write of y at "ompScalar.cpp":65 conflicts with a prior memory write of y at "ompScalar.cpp":65 (output dependence)	1
1	5	Write -> Read data-race		Memory read of x at "ompScalar.cpp":71 conflicts with a prior memory write of x at "ompScalar.cpp":64 (flow dependence)	100
1	6	Read -> Write data-race		Memory write of x at "ompScalar.cpp":64 conflicts with a prior memory read of x at "ompScalar.cpp":71 (anti dependence)	100
1	7	Write -> Read data-race		Memory read of y at "ompScalar.cpp":71 conflicts with a prior memory write of y at "ompScalar.cpp":65 (flow dependence)	100
1	8	Read -> Write data-race		Memory write of y at "ompScalar.cpp":65 conflicts with a prior memory read of y at "ompScalar.cpp":71 (anti dependence)	100
2	9	OpenMP -- cannot be private		OpenMP -- The access at "ompScalar.cpp":71 cannot be private because it expects the value previously defined at "ompScalar.cpp":71 in the serial execution	99
3	10	OpenMP -- undefined in the serial code (original program)		OpenMP -- undefined in the serial code (original program) at "ompScalar.cpp":77 with "ompScalar.cpp":71	1
3	11	OpenMP -- undefined in the serial code (original program)		OpenMP -- undefined in the serial code (original program) at "ompScalar.cpp":77 with "ompScalar.cpp":50	1
4	12	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 6. Результат анализа параллельной реализации 2 скалярного умножения

Первое, что можно отметить – появились новые типы сообщений. Вернемся к этому позже. Второй интересный момент – ошибка за номером 1.

«Открываем» ее двойным щелчком:

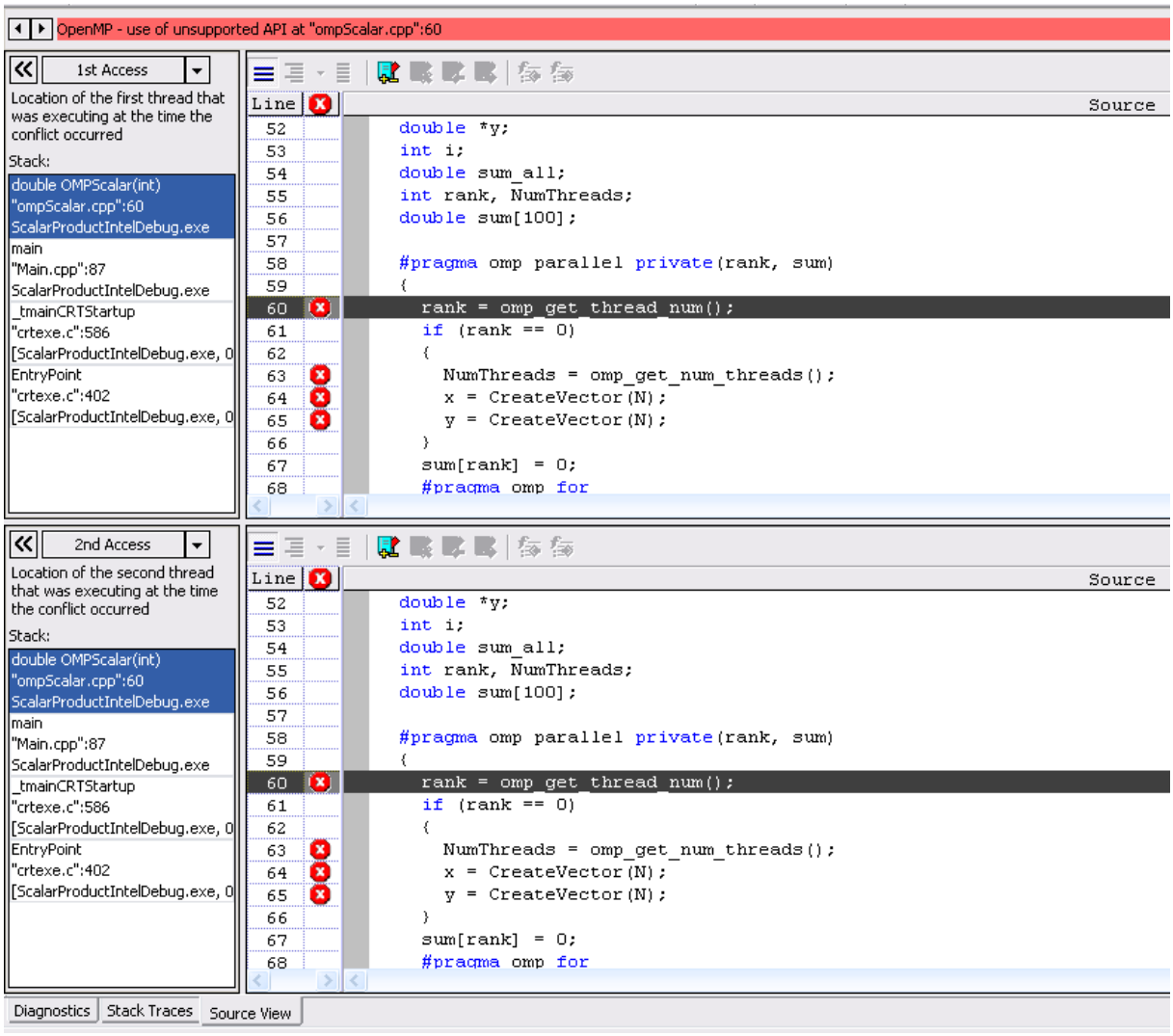


Рис. 7. Просмотр диагностики «Use of unsupported API»

Представленная диагностика вызвана тем, что ITC «не понимает» run-time функции OpenMP и в частности **`omp_get_num_threads()`**. Отсюда, кстати, и большая часть остальных ошибок, которые вызваны тем, что значение переменной **`rank`** не известно ITC, а соответственно, он находит конфликты там, где их на самом деле нет.

Выходов из подобных ситуаций два: отказаться от использования run-time функций OpenMP (тем более, что их использование при программировании на OpenMP считается чем-то вроде плохого тона) или отказаться от компиляторного инструментирования, то есть удалить из опций проекта ключ /Qtcheck.

Но прежде вернемся к ошибке под номером 9.

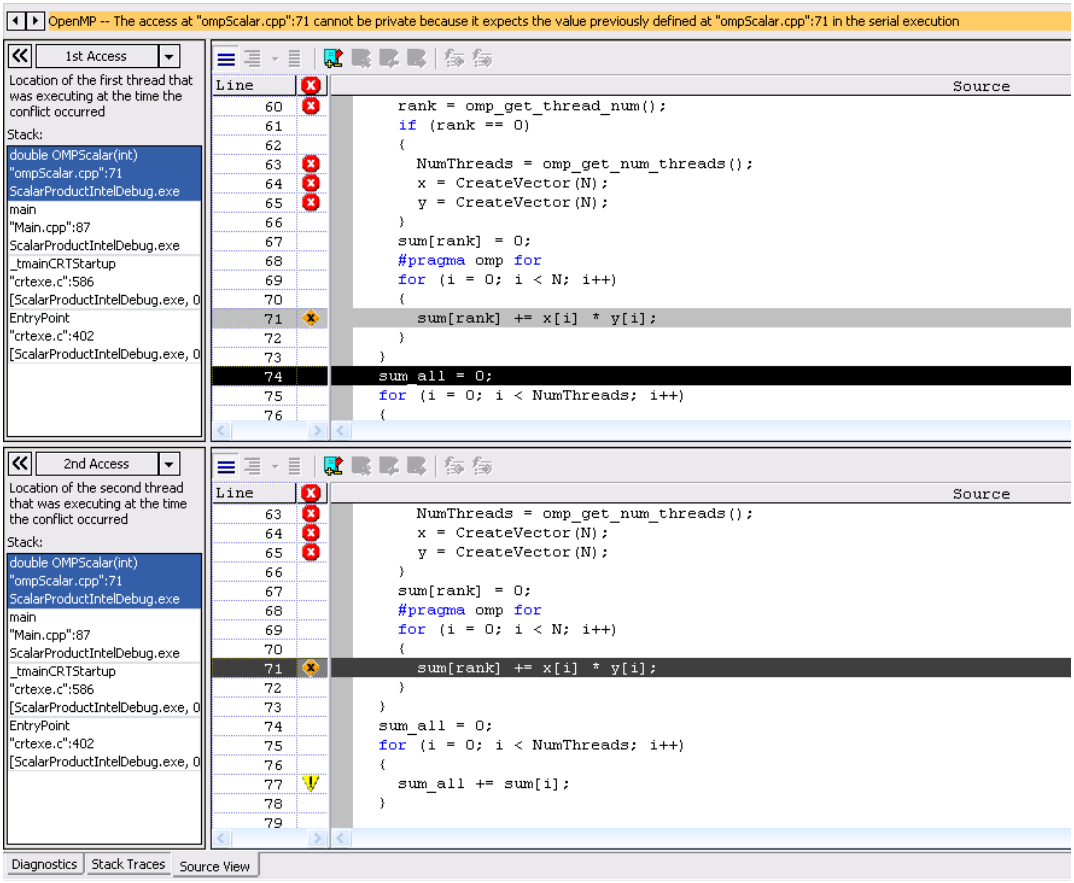


Рис. 8. Неверная локализация переменной sum

Несмотря на не слишком вразумительное сообщение, тем не менее, можно понять, что выше по коду нами допущена ошибка – переменная **sum** сделана **private**, что, конечно же, неверно. Этот массив специально был создан для подсчета потоками локальных сумм и естественно должен быть общим, чтобы по окончании параллельной секции из него в переменную **sum_all** можно было собрать итоговую сумму.

Исправляем ситуацию:

```
#pragma omp parallel private(rank/*, sum*/)
{
  ...
}
sum_all = 0;
for (i = 0; i < NumThreads; i++)
{
  sum_all += sum[i];
}

DeleteVector(x);
DeleteVector(y);
```

Убираем ключ /Qtcheck из проекта и снова запускаем анализ в ITC.

Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Read data-race**		**Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "ompScalar.cpp":103 (flo...**	**50**
1	2	Write -> Read data-race		Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "Vector.cpp":29 (flow dependence)	50
1	3	Write -> Read data-race		Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "ompScalar.cpp":104 (flow dependence)	50
1	4	Write -> Read data-race		Memory read at "ompScalar.cpp":110 conflicts with a prior memory write at "Vector.cpp":29 (flow dependence)	50
2	5	Thread termination		Thread termination at "ompScalar.cpp":97 - includes stack allocation of 3, MB and use of 4, KB	1
3	6	Thread termination		Thread termination at "ompScalar.cpp":97 - includes stack allocation of 1, MB and use of 4, KB	1
4	7	Thread termination		Thread termination at "Main.cpp":50 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 9. Результат анализа параллельной реализации 2 после первого исправления

Теперь мы, к сожалению, потеряли информацию об именах переменных. Число сообщений также сократилось. Однако ITC все еще видит в коде ошибки.

Анализируя первое же сообщение, мы находим одну из типичных ошибок в многопоточных программах – использование потоками неинициализированных данных.

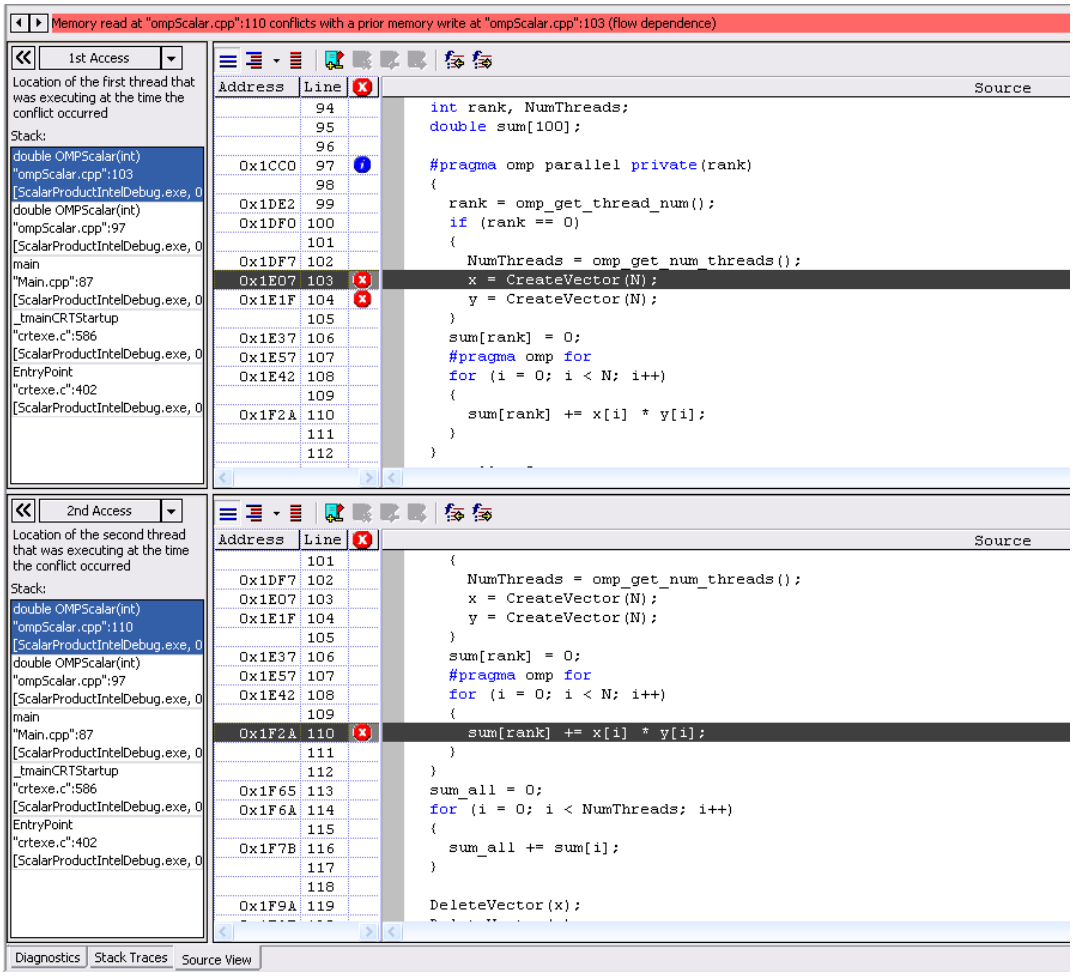


Рис. 10. Ошибка – использование потоками неинициализированных данных

Глядя на выделенные строки, нетрудно понять, что поток с номером, отличным от 0, будет пытаться использовать переменные **x** и **y** до того, как нулевой поток выделит под них память, что в лучшем случае приведет к падению, а в худшем к неверному результату расчетов. Решение в данной ситуации также является типичным и состоит в установке барьера после участка инициализации.

```
#pragma omp parallel private(rank)
```

```
{
  rank = omp_get_thread_num();
  if (rank == 0)
  {
    NumThreads = omp_get_num_threads();
    x = CreateVector(N);
    y = CreateVector(N);
  }
  sum[rank] = 0;
  #pragma omp barrier
  #pragma omp for
  for (i = 0; i < N; i++)
  {
    sum[rank] += x[i] * y[i];
  }
}
...
```

После внесения последних исправлений, результаты запуска полученной параллельной версии, наконец-то, совпадут с последовательным вариантом. Убедитесь в этом, а также в том, что ITC подтвердит отсутствие проблем в коде.

2.2. Задача Дирихле

2.2.1. Постановка задачи

В качестве второго примера рассмотрим задачу из области численного решения дифференциальных уравнений в частных производных, а именно задачу Дирихле для уравнения Пуассона (см., например, [8]). Итак, необходимо найти функцию $u = u(x, y)$, удовлетворяющую в области определения D уравнению

$$\begin{cases} \dfrac{\delta^2 u}{\delta x^2} + \dfrac{\delta^2 u}{\delta y^2} = f(x,y), & (x,y) \in D, \\ u(x,y) = g(x,y), & (x,y) \in D^0, \end{cases}$$

и принимающую на границе D^0 области D значения $g(x, y)$.

2.2.2. Метод решения

Используя распространенный метод конечных разностей (он же метод сеток) перепишем уравнение Пуассона в конечно-разностной форме [9],

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

Разрешив его относительно u_{ij}, получим

$$u_{ij} = 0.25 \ \ (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Мы получили основу для построения итерационной схемы решения задачи Дирихле, в которой, отталкиваясь от некоторого начального приближения можно последовательно уточнять значения u_{ij} до достижения требуемой точности.

2.2.3. Последовательная реализация

Последовательная реализация данной итерационной схемы может выглядеть следующим образом:

```
do
{
```

```
  dmax = 0;
  for (i = 1; i < N - 1; i++)
  {
    for(j = 1; j < N - 1; j++)
    {
      temp = u[N * i + j];
      u[N * i + j] = 0.25 * (u[N * i + j + 1] + u[N * i + j - 1] +
        u[N * (i + 1) + j] + u[N * (i - 1) + j]);
      dm = fabs(u[N * i + j] - temp);
      if (dmax < dm)
        dmax = dm;
    }
  }
}
while (dmax > EPS);
```

Здесь **dmax** есть максимальная разность между «старым», с предыдущей итерации, и «новым», посчитанным на текущей, значениями u_{ij} и используется для принятия решения об окончании расчетов.

2.2.4. Параллельная реализация, вариант 1

«Сеточные» задачи, в которых решение получается выполнением набора из одних и тех же действий в каждом «узле» сетки обладают очень простой схемой распараллеливания. Достаточно сетку «порезать» на части: вертикально на столбцы, горизонтально на полосы, или и так и так, то есть на блоки, и распараллеливание выполнено. Если при этом объем вычислений, выполняемых в каждом узле, примерно одинаков, то и эффективность полученной параллельной реализации может быть довольно высока.

Таким образом, в данной работе мы рассмотрим две схемы распараллеливания решения задачи Дирихле. Первая – схема с максимальным параллелизмом, в которой сетка «режется» на блоки размера 1 на 1. Вторая – схема, в которой распределение вычислений осуществляется по строкам сетки.

В данной лабораторной работе нашей целью не является производительность получаемых параллельных реализаций, поэтому в качестве первого варианта, как было сказано выше, мы используем вариант с максимальным параллелизмом, который в OpenMP версии достигается распараллеливанием обоих циклов:

```
do
{
  dmax = 0;
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++)
  {
    #pragma omp parallel for
    for(j = 1; j < N - 1; j++)
    {
      temp = u[N * i + j];
      u[N * i + j] = 0.25 * (u[N * i + j + 1] + u[N * i + j - 1] +
        u[N * (i + 1) + j] + u[N * (i - 1) + j]);
      dm = fabs(u[N * i + j] - temp);
      if (dmax < dm)
        dmax = dm;
    }
  }
}
while (dmax > EPS);
```

Конечно же, в представленном виде код является некорректным. В чем именно, нам предстоит выяснить, используя Intel Thread Checker.

2.2.5. Анализ реализации 1

Исследуем представленный выше код на наличие ошибок.

Для проверки правильности работы параллельной программы можно применить, казалось бы, естественный подход – сравнить результаты выполнения последовательной и параллельной версий. Например, в задаче умножения матриц результат должен быть один и тот же, независимо от применяемой вычислительной схемы. Вместе с тем, часто наблюдается и иная картина – именно так, как в рассматриваемом примере – результаты параллельной версии могут отличаться от результатов последовательной, но это не будет являться признаком наличия ошибок. Одна из причин различия результатов – изменения порядка выполнения операций вещественной арифметики, что может, в частности, привести к изменению величины получаемой погрешности вычислений. Ситуация усложняется, если вычислительная схема параллельного алгоритма отличается от исходного последовательного прототипа – так, в нашем примере, порядок обработки узлов вычислительной сетки в параллельной версии алгоритма может быть другим нежели в последовательном методе. Более того, результаты параллельных расчетов могут не совпадать при различных запусках даже при одних и тех же начальных данных, поскольку условия запуска (например, загрузка вычислительной системы) также могут влиять на порядок вычислений. Описанная проблема – различие выполняемых вычислений – является одной из принципиальных при разработке параллельных программ. Наличие такой проблемы приводит к тому, что при разработке параллельного метода необходимо тщательно анализировать идентичность вычислительных схем параллельных и последовательных расчетов, а при обнаружении различий доказывать корректность параллельной версии и определять способы проверки правильности выполнения параллельных расчетов (дополнительная информация по данному вопросу может быть получена, например, в материале Лекции 11 в [7]).

Рассматриваемый пример прекрасно демонстрирует описанные выше ситуации – вычислительная схема параллельного алгоритма отличается от последовательного метода (порядок обработки узлов вычислительной сетки может оказаться различным) и результаты параллельных вычислений могут отличаться от запуска к запуску. В теоретических работах показана корректность рассмотренной параллельный схемы вычислений – процесс вычислений сходится к решению поставленной задачи Дирихле. Данный математический результат дает подход для проверки правильности выполнения параллельной программы путем оценки точности получения требуемого результата вычислений (в силу сходимости параллельный алгоритм должен обеспечивать достижение любой наперед заданной точности расчетов). Иными словами, для проверки правильности можно сравнить результат параллельной программы с точным решением и, если результат совпадает с точным решением в пределах задаваемой точностью, то можно считать, что расчеты выполнены правильно. При этом, однако, следует понимать, что такая проверка не дает полной гарантии – как и любое тестирование, правильность прохождения теста говорит о корректности работы программы только при данных исходных параметрах задачи (а в случае параллельных вычислений – о корректности только данного конкретного запуска программы). Необходимы многократные запуски при одних и тех же исходных данных, нужны различные варианты постановок решаемой задачи. Процесс тестирования становится трудоемким и длительным – все отмеченные моменты убедительно подтверждают необходимость использования инструментов отладки параллельных программ.

Проанализируем код с помощью Intel Thread Checker.

Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Write data-race**		**Memory write of temp at "ompGZ.cpp":39 conflicts with a prior memory write of temp at "ompGZ.cpp":39 (output dependence)**	**20273**
1	2	Read -> Write data-race		Memory write of temp at "ompGZ.cpp":39 conflicts with a prior memory read of temp at "ompGZ.cpp":42 (anti...	1986754
1	3	Write -> Read data-race		Memory read of v[] at "ompGZ.cpp":40 conflicts with a prior memory write of v[] at "ompGZ.cpp":40 (flow dependence)	1986754
1	4	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":40 conflicts with a prior memory read of v[] at "ompGZ.cpp":40 (anti dependence)	1986754
1	5	Write -> Write data-race		Memory write of dm at "ompGZ.cpp":42 conflicts with a prior memory write of dm at "ompGZ.cpp":42 (output...	20273
1	6	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":42 conflicts with a prior memory read of dm at "ompGZ.cpp":45 (anti dependence)	9506
1	7	Write -> Read data-race		Memory read of dmax at "ompGZ.cpp":44 conflicts with a prior memory write of dmax at "ompGZ.cpp":45 (flow...	1846285
1	8	Write -> Write data-race		Memory write of dmax at "ompGZ.cpp":45 conflicts with a prior memory write of dmax at "ompGZ.cpp":45 (output dependence)	3001
1	9	Read -> Write data-race		Memory write of dmax at "ompGZ.cpp":45 conflicts with a prior memory read of dmax at "ompGZ.cpp":44 (anti...	16309
1	10	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":42 conflicts with a prior memory read of dm at "ompGZ.cpp":44 (anti dependence)	1977248
2	11	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 11. Задача Дирихле – результат анализа параллельной реализации 1

Как видим, налицо существенное количество ошибок, которые ITC классифицирует как «гонка данных».

«Разворачивая» любую из них, мы получим информацию вида:

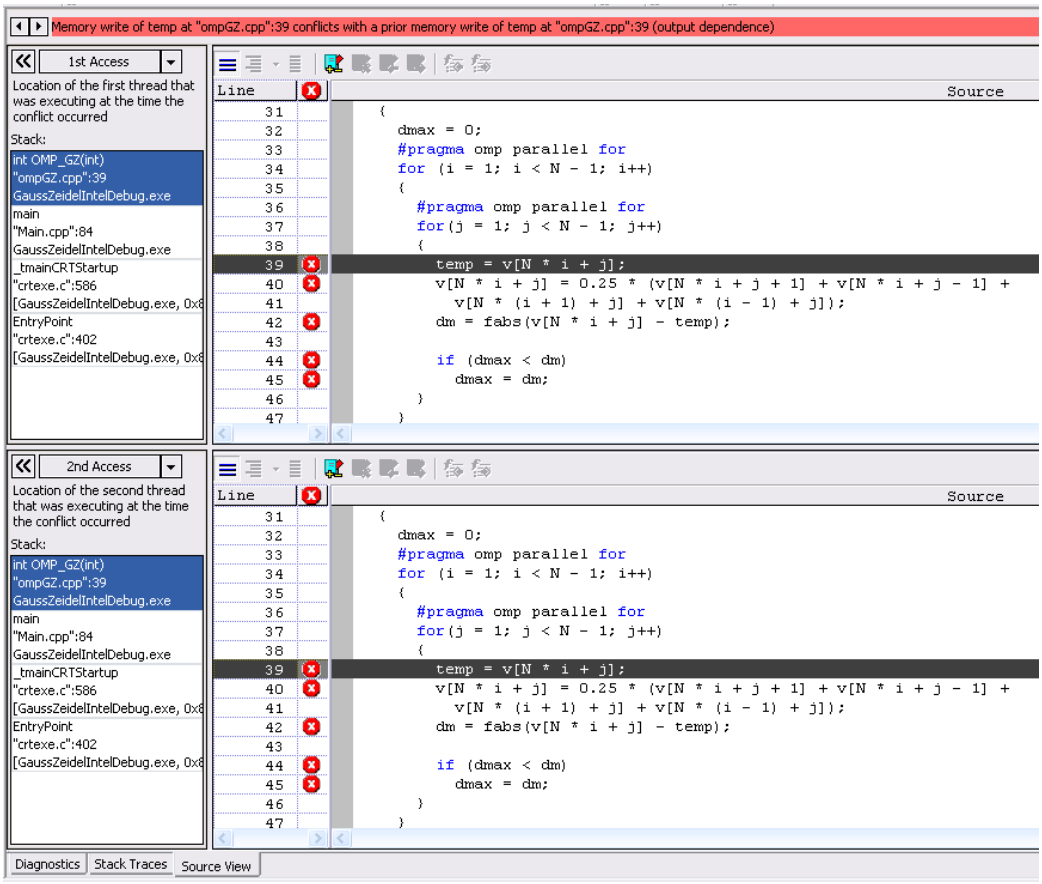


Рис. 12. Задача Дирихле – ошибка типа «гонка данных»

Как видим, диагностика ITC указывает нам и строку, в которой имеет место проблема, и переменную, с которой она связана. В выделенной строке обнаружен конфликт доступа к переменной **`temp`**, когда два потока могут пытаться одновременно записать в нее значения.

Решение выявленных проблем зависит от того, как используется та или иная переменная и может состоять в ее «локализации», то есть создании внутренней для каждого потока копии, как для переменной **`temp`**, или в синхронизации доступа к ней, если переменная нужна всем потокам, как в случае с **`dmax`**.

Заметим, что две из найденных ITC ошибок в данной реализации могут проявиться только при очень маленьком значении *N*. Какие именно, предлагаем читателям найти самостоятельно.

Корректируем код:

```
omp_lock_t dmax_lock;

omp_init_lock(&dmax_lock);
do
{
  dmax = 0;
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++)
  {
    #pragma omp parallel for private(temp, dm)
    for(j = 1; j < N - 1; j++)
    {
      temp = v[N * i + j];
      v[N * i + j] = 0.25 * (v[N * i + j + 1] + v[N * i + j - 1] +
        v[N * (i + 1) + j] + v[N * (i - 1) + j]);
      dm = fabs(v[N * i + j] - temp);

      omp_set_lock(&dmax_lock);
      if (dmax < dm)
        dmax = dm;
      omp_unset_lock(&dmax_lock);
    }
  }
}
while (dmax > EPS);

omp_destroy_lock(&dmax_lock);
```

По результатам запуска убеждаемся, что параллельная версия работает корректно. Снова запускаем Thread Checker и видим.

Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Read data-race**		**Memory read of v[] at "ompGZ.cpp":138 conflicts with a prior memory write of v[] at...**	**198675**
1	2	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":138 conflicts with a prior memory read of v[] at "ompGZ.cpp":138 (anti dependence)	1986754
2	3	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 13. Задача Дирихле – результат анализа параллельной реализации 1 после исправления

Прав или нет ITC в своих подозрениях относительно последнего варианта кода предлагаем читателям выяснить самостоятельно.

2.2.6. Параллельная реализация, вариант 2

Второй вариант распараллеливания, который мы будем использовать в данной работе, состоит в уменьшении степени параллелизма реализации, что, тем не менее, ведет к увеличению эффективности. Вариант заключается в распараллеливании только внешнего цикла и дополнительно предполагает некоторые изменения в схеме подсчета значения **`dmax`**:

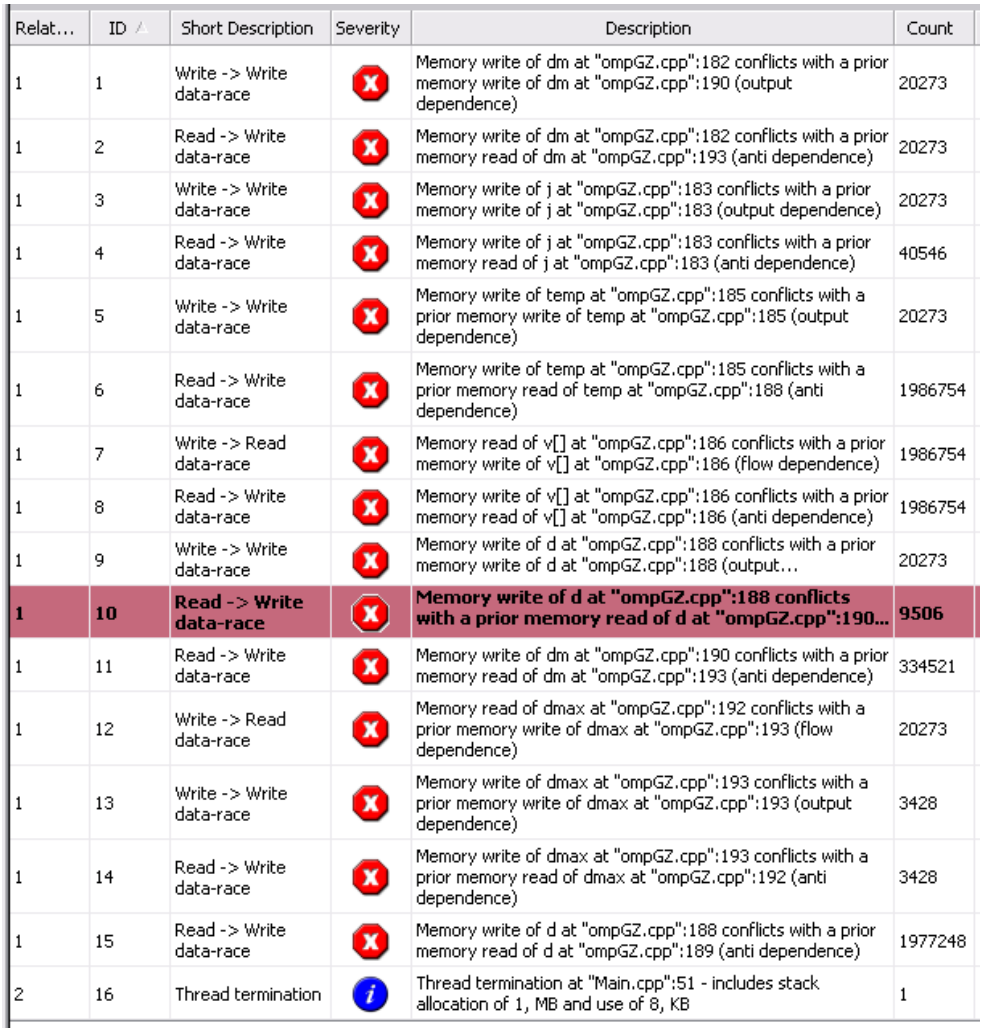
```
do
{
  dmax = 0;
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++)
  {
    dm = 0;
    for(j = 1; j < N - 1; j++)
    {
      temp = u[N * i + j];
      u[N * i + j] = 0.25 * (u[N * i + j + 1] + u[N * i + j - 1] +
        u[N * (i + 1) + j] + u[N * (i - 1) + j]);
      d = fabs(u[N * i + j] - temp);
      if (dm < d)
        dm = d;
    }
    if (dmax < dm)
      dmax = dm;
  }
}
while (dmax > EPS);
```

Как и в первом случае представленный код содержит ошибки, которые необходимо найти с помощью Intel Thread Checker и устранить.

2.2.7. Анализ реализации 2

Прежде всего, отметим, что в связи с изменившейся схемой работы и рассмотренная выше параллельная версия и данная могут не давать идентичные результаты по сравнению с последовательным кодом, что, конечно, является дополнительным усложняющим моментом для «ручной» отладки.

Проанализируем код с помощью Intel Thread Checker.



Relat...	ID	Short Description	Severity	Description	Count
1	1	Write -> Write data-race		Memory write of dm at "ompGZ.cpp":182 conflicts with a prior memory write of dm at "ompGZ.cpp":190 (output dependence)	20273
1	2	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":182 conflicts with a prior memory read of dm at "ompGZ.cpp":193 (anti dependence)	20273
1	3	Write -> Write data-race		Memory write of j at "ompGZ.cpp":183 conflicts with a prior memory write of j at "ompGZ.cpp":183 (output dependence)	20273
1	4	Read -> Write data-race		Memory write of j at "ompGZ.cpp":183 conflicts with a prior memory read of j at "ompGZ.cpp":183 (anti dependence)	40546
1	5	Write -> Write data-race		Memory write of temp at "ompGZ.cpp":185 conflicts with a prior memory write of temp at "ompGZ.cpp":185 (output dependence)	20273
1	6	Read -> Write data-race		Memory write of temp at "ompGZ.cpp":185 conflicts with a prior memory read of temp at "ompGZ.cpp":188 (anti dependence)	1986754
1	7	Write -> Read data-race		Memory read of v[] at "ompGZ.cpp":186 conflicts with a prior memory write of v[] at "ompGZ.cpp":186 (flow dependence)	1986754
1	8	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":186 conflicts with a prior memory read of v[] at "ompGZ.cpp":186 (anti dependence)	1986754
1	9	Write -> Write data-race		Memory write of d at "ompGZ.cpp":188 conflicts with a prior memory write of d at "ompGZ.cpp":188 (output...	20273
1	**10**	**Read -> Write data-race**		**Memory write of d at "ompGZ.cpp":188 conflicts with a prior memory read of d at "ompGZ.cpp":190...**	**9506**
1	11	Read -> Write data-race		Memory write of dm at "ompGZ.cpp":190 conflicts with a prior memory read of dm at "ompGZ.cpp":193 (anti dependence)	334521
1	12	Write -> Read data-race		Memory read of dmax at "ompGZ.cpp":192 conflicts with a prior memory write of dmax at "ompGZ.cpp":193 (flow dependence)	20273
1	13	Write -> Write data-race		Memory write of dmax at "ompGZ.cpp":193 conflicts with a prior memory write of dmax at "ompGZ.cpp":193 (output dependence)	3428
1	14	Read -> Write data-race		Memory write of dmax at "ompGZ.cpp":193 conflicts with a prior memory read of dmax at "ompGZ.cpp":192 (anti dependence)	3428
1	15	Read -> Write data-race		Memory write of d at "ompGZ.cpp":188 conflicts with a prior memory read of d at "ompGZ.cpp":189 (anti dependence)	1977248
2	16	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 14. Задача Дирихле – результат анализа параллельной реализации 2

Количество ошибок в этой версии еще больше в связи с увеличившимся количеством переменных, кроме того, в список попала переменная **`j`**, заботу о которой раньше «брал не себя» компилятор – как известно переменную цикла в директиве **`#pragma omp parallel for`** не обязательно объявлять как **`private`**.

Также как и для варианта 1 исправление найденных ошибок заключается в локализации необходимых переменных и использовании синхронизации при работе с переменной **`dmax`**.

```
omp_lock_t dmax_lock;

omp_init_lock(&dmax_lock);
do
{
  dmax = 0;
  #pragma omp parallel for private(j, temp, d, dm)
  for (i = 1; i < N - 1; i++)
  {
    dm = 0;
    for(j = 1; j < N - 1; j++)
    {
      temp = v[N * i + j];
      v[N * i + j] = 0.25 * (v[N * i + j + 1] + v[N * i + j - 1] +
        v[N * (i + 1) + j] + v[N * (i - 1) + j]);
      d = fabs(u[N * i + j] - temp);
      if (dm < d)
        dm = d;
    }
```

```
    omp_set_lock(&dmax_lock);
    if (dmax < dm)
      dmax = dm;
    omp_unset_lock(&dmax_lock);
  }
}
while (dmax > EPS);

omp_destroy_lock(&dmax_lock);
```

Снова запускаем ITC.

Re...	ID	Short Description	Severity	Description	Count
1	1	Write -> Read data-race		Memory read of v[] at "ompGZ.cpp":233 conflicts with a prior memory write of v[] at...	198675
1	2	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":233 conflicts with a prior memory read of v[] at "ompGZ.cpp":233 (anti dependence)	1986754
2	3	Thread termination		Thread termination at "Main.cpp":51 - includes stack allocation of 1, MB and use of 8, KB	1

Рис. 15. Задача Дирихле – результат анализа параллельной реализации 2 после исправления

Как и выше ITC нашел проблему там, где ее на самом деле нет. Почему это так, предлагаем читателям выяснить самостоятельно. В качестве подсказки: вспомните характер распределения итераций цикла **for** при использовании директивы **#pragma omp parallel for** без дополнительных параметров.

2.3. Задача об обедающих философах

На примере данной классической задачи мы продемонстрируем еще одну типичную ошибку – *тупик* (*deadlock*), ее диагностику при помощи ITC и один из способов устранения.

2.3.1. Постановка задачи

Приведем вольную формулировку задачи Дейкстры: за круглым столом заседают 5 философов. Напротив каждого из них стоит блюдо со спагетти. Между каждыми двумя соседями расположена одна вилка. Философ может находиться в одном из двух состояний: ест, размышляет. При еде философу нужны 2 вилки (левая и правая). Стратегия поведения философа следующая: он берет левую вилку (если она свободна) и затем, дождавшись правой вилки, начинает есть. Поев, он освобождает вилки в обратном порядке. Реализовать симулятор, демонстрирующий заседание философов.

2.3.2. Параллельная реализация, вариант 1

Реализуем требуемый симулятор на основе потоков Windows Threads.

Идея реализации состоит в следующем: главный поток создает дополнительные потоки в соответствии с количеством философов, запускает их и переходит в бесконечный цикл. Каждый из дополнительных потоков реализует поведение философа. При этом вилки предлагается моделировать при помощи мьютексов, обеспечивая тем самым процедуру «захвата вилки» философом. Функция потока будет принимать в качестве параметров структуру, содержащую мьютексы, соответствующие левой и правой вилкам, а также порядковый номер философа.

Сделаем следующие объявления:

```
// Количество философов
const unsigned int n = 5;

// Структура - описание философа
typedef struct
{
  int iID;                                    // Номер философа
  HANDLE hMyObjects[2];                       // Мьютексы (вилки)
} THREADCONTROLBLOCK, *PTHREADCONTROLBLOCK;
```

Приведем функцию потока. Используем функцию **WaitForSingleObject** для ожидания освобождения ресурса (мьютекса).

```
long WINAPI ThreadRoutine(long lParam)
{
  PTHREADCONTROLBLOCK pcb=(PTHREADCONTROLBLOCK)lParam;
  while (TRUE)
  {
    WaitForSingleObject(pcb->hMyObjects[0],INFINITE);
    WaitForSingleObject(pcb->hMyObjects[1],INFINITE);
    printf("Eating: Philosopher %d \n",pcb->iID);
    ReleaseMutex(pcb->hMyObjects[1]);
    ReleaseMutex(pcb->hMyObjects[0]);
  };
  return (0);
}
```

Тогда функция **`main`** будет выглядеть так:

```
int main()
{
  HANDLE hMutexes[n];
  THREADCONTROLBLOCK tcb[n];
  int iThreadID;

  for (int i = 0; i < n; i++)
    hMutexes[i] = CreateMutex(NULL, FALSE, NULL);

  for (int i = 0; i < n; i++)
  {
    tcb[i].iID = i+1;
    tcb[i].hMyObjects[0] = hMutexes[i % n];
    tcb[i].hMyObjects[1] = hMutexes[(i+1) % n];
    CloseHandle(CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadRoutine,
      (void *)&tcb[i],0,LPDWORD(&iThreadID)));
   }
   while(TRUE);
   return(0);
}
```

2.3.3. Анализ реализации 1

Запустим программу на выполнение несколько раз.

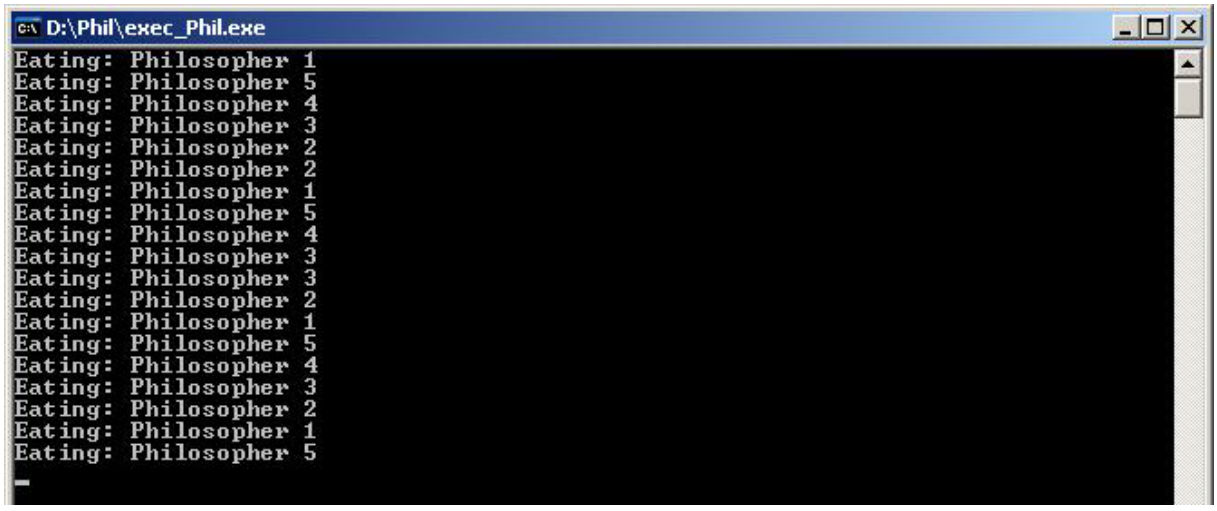


Рис. 16. Задача об обедающих философах – результаты запуска параллельной реализации 1

Результаты будут варьироваться от запуска к запуску. Единственное, что их объединяет, – неизменное зависание в некоторый момент. Попробуем разобраться, в чем дело. Прибегнем к помощи Intel Thread Checker. Как видно из рисунка, ITC сгенерировал 5 диагностических сообщений,

представляющих для нас интерес. Каждое из сообщений соответствует одному из созданных нами потоков и «говорит» о наличии тупика.

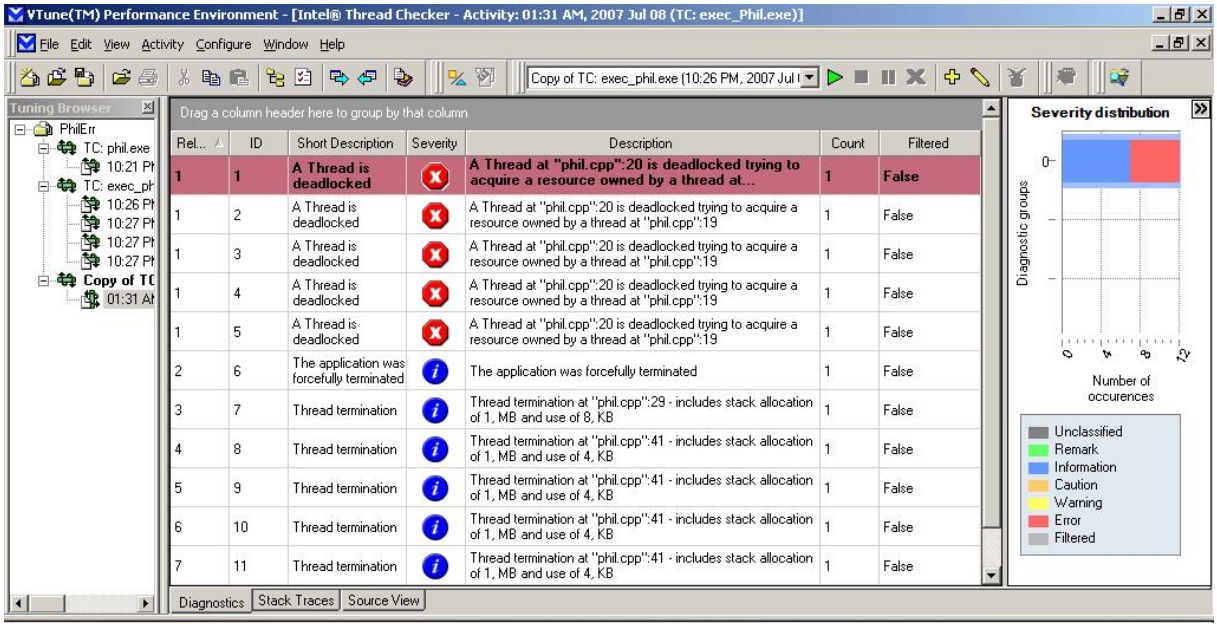


Рис. 17. Диагностика ITC в задаче об обедающих философах (параллельная реализация 1)

2.3.4. Параллельная реализация, вариант 2

Подумаем над тем, как исключить тупики, наличие которых обуславливает не столько некорректная реализация, сколько сама постановка задачи. Действительно, возможна ситуация, в которой каждый из философов взял ровно одну вилку и ждет, когда освободится вторая, которая занята соседом. Сосед в свою очередь ждет свою вторую вилку и т.д.

Одним из возможных способов решения проблемы является изменение модели поведения философа. К примеру, можно наделить его обязанностью брать вилки одновременно, лишь тогда, когда обе они свободны. Изменения в программной реализации будут минимальны – достаточно заменить 2 вызова функции **`WaitForSingleObject`** на 1 вызов функции **`WaitForMultipleObjects`** для одновременного захвата мьютексов, соответствующим обеим вилкам.

```
#include <stdio.h>
#include <windows.h>

// Количество философов
const unsigned int n = 5;

typedef struct {
  int iID;
  HANDLE hMyObjects[2];
} THREADCONTROLBLOCK, *PTHREADCONTROLBLOCK;

long WINAPI ThreadRoutine(long lParam) {
  PTHREADCONTROLBLOCK pcb=(PTHREADCONTROLBLOCK)lParam;
  while (TRUE) {
    WaitForMultipleObjects(2, pcb->hMyObjects, TRUE, INFINITE);
    printf("Eating: Philosopher %d \n",pcb->iID);
    ReleaseMutex(pcb->hMyObjects[1]);
    ReleaseMutex(pcb->hMyObjects[0]);
  };
return (0);
}

int main() {
   HANDLE hMutexes[n];
```

```
   THREADCONTROLBLOCK tcb[n];
   int iThreadID;

   for (int i = 0; i < n; i++)
     hMutexes[i] = CreateMutex(NULL,FALSE,NULL);

   for (int i = 0; i < n; i++) {
     tcb[i].iID = i+1;
     tcb[i].hMyObjects[0] = hMutexes[i % n];
     tcb[i].hMyObjects[1] = hMutexes[(i+1) % n];
     CloseHandle(CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadRoutine,
                         (void *)&tcb[i],0,LPDWORD(&iThreadID)));
   }
   while(TRUE);
   return(0);
}
```

2.3.5. Анализ реализации 2

Тестовые запуски подтверждают наши ожидания – программа перестала зависать. ITC также не делает по представленному выше коду никаких замечаний.

2.4. Задача о роботе[2]

2.4.1. Постановка задачи

Для постановки задачи рассмотрим некоторую «гипотетическую» лабораторию искусственного интеллекта, в которой выполняются работы по созданию роботов.

Для испытаний лабораторных образцов построен полигон, устроенный следующим образом: в начале полигона находится единственная дверь. Пройдя через нее, робот оказывается перед B дверьми. Выбрав одну из дверей, робот вновь оказывается перед B дверьми, и т.д. Пройдя через дверь, робот получает премию в размере x, где x – количество монет, лежащее за данной дверью (для разных дверей количество монет может быть разным).

Считая известным:

- количество дверей (B),
- количество уровней полигона (L),
- количество монет лежащее за дверью i на уровне L (x_L^i) и
- вероятности выбора роботом, находящемся за дверью i на уровне L, двери j на уровне $L+1$ ($P_{L,L+1}^{i,j}$),

следует определить среднюю премию, получаемую роботом при проходе через данный полигон.

Считать, что выполняются следующие условия: $B \le 20, L \le 10$.

2.4.2. Модель

Объектом исследования в данной задаче является полигон с иерархической структурой, по которому по определенному алгоритму перемещается робот. Построим математическую модель исследуемого объекта.

Прежде всего, введем необходимые обозначения.

- B – количество вариантов пути, выбираемых роботом на каждом шаге.
- L – количество уровней полигона.

[2] Данная задача при выполнении лабораторной работы рассматривается как дополнительное задание

- $P_{L,L+1}^{i,j}$ – вероятность попадания из узла *i* уровня *L* в узел *j* уровня *L+1*.
- x_L^i – количество монет, лежащее за дверью *i* уровня *L*.

B, *L*, $P_{L,L+1}^{i,j}$ и x_L^i являются исходными данными для данной задачи. Из условия задачи вытекают следующие ограничения на исходные данные:

- $P_{L,L+1}^{i,j} = 0$, если двери *i* и *j* не соединены.
- $\sum_{j=0}^{B-1} P_{L,L+1}^{i,j} = 1$.

Теперь перейдем непосредственно к выбору модели. Вследствие иерархической структуры полигона выглядит разумным его представление в виде дерева степени *B* и глубины *L*. При этом узлы дерева содержат значения x_L^i, а дугам приписаны вероятности перехода $P_{L,L+1}^{i,j}$.

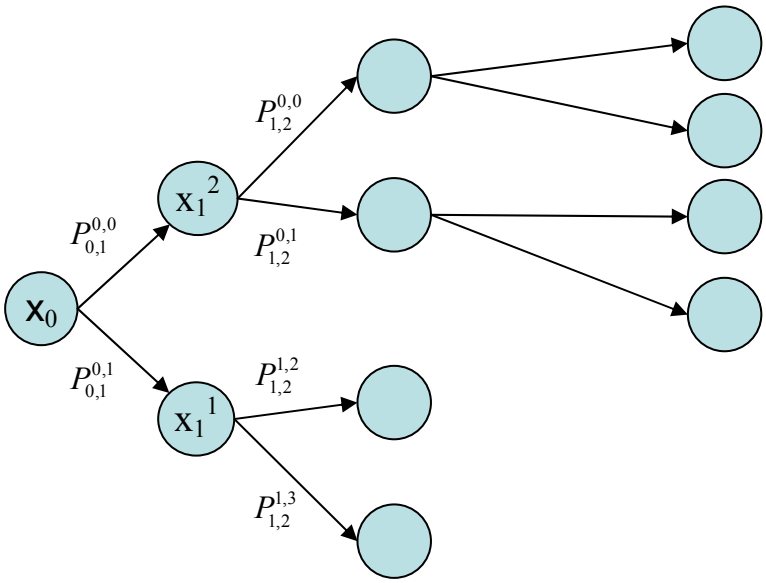


Рис. 18. Модель полигона в задаче о роботе

2.4.3. Метод решения

Перейдем к описанию метода нахождения результата. Пусть E_d^i – средняя премия, которая может быть получена, если робот начинает путь в узле *i* уровня *d*.

Тогда

$$E_d^i = x_d^i + \sum_{j=0}^{B-1} P_{d,d+1}^{i,j} \times E_{d+1}^j \quad , \text{ где } d = \overline{0; l-2};\ i = \overline{0; B-1}.$$

$$E_{L-1}^i = x_{L-1}^i, \text{ где } i = \overline{0; B-1}.$$

Очевидный метод вычисления результата E_0^0 состоит в вычислении значений *E* на последнем уровне дерева (второе соотношение) с последующим пересчетом из конца в начало (первое рекуррентное соотношение).

2.4.4. Последовательная реализация

Перейдем к реализации последовательной версии изложенного выше метода решения.

Проектирование структур данных

Предусмотрим для представления дерева константы, хранящие степень *B* и глубину *L*. Для организации дерева создадим структуру, соответствующую узлу дерева. Будем хранить в этой структуре количество монет *x* и вероятность *p* попадания в узел из узла предыдущего уровня. Учитывая метод решения, состоящий в последовательном пересчете рекуррентных соотношений из пункта 2.4.3, добавим в рассматриваемую структуру специальное поле для хранения текущего значения средней премии *E*. В результате получим следующие объявления:

```
const int b = 10;          // Степень дерева – B в постановке задачи
const int l = 7;           // Количество уровней дерева

const int MAX_VALUE = 20; // Максимальное значение в вершине дерева

int NodesCount;            // Количество узлов дерева = (1-b^l) / (1-b)
int BranchesCount;         // Количество ветвей дерева = NodesCount - 1;

typedef struct             // Узел дерева
{
  int value;               // Значение в узле
  // Вероятность попадания в узел из узла предыдущего уровня
  double probability;
  // Компонент для подсчета среднего
  double expectation;
} TreePart;
```

Учитывая регулярную структуру дерева, будем хранить его в виде массива узлов, располагая в нем узлы последовательно по уровням, от корня к листьям. Учитывая возможный большой размер массива, будем создавать его динамически в куче. В результате получим:

```
TreePart *RobotTree;  // Массив для хранения дерева.
// Схема хранения:
// (V00)
// (V10 V11 ... V1b)
// (V21 V22... V2b^2)
// ...
// (Vl-1,1 Vl-1,2...Vl-1,b^(l-1)),
// где скобки стоят для наглядности.
// Дерево упаковывается в одномерный массив по уровням.
```

Для удобства индексации и снижения накладных расходов предусмотрим однократное вычисление значения *b* в степени от 0 до l-1, а также номеров первых узлов каждого уровня дерева в массиве RobotTree.

```
// Вспомогательный массив для хранения значений b в степени 0...l-1
__int64 power[l];
// Вспомогательный массив для хранения номера первого узла каждого уровня
//  в массиве RobotTree
__int64 index[l];
```

Проектирование модульной структуры

Предусмотрим наличие в проекте файла **robot.cpp**, содержащего рассмотренные выше объявления, а также следующих функций:

```
// Ввод дерева
void InputTree(void);
// Освобождение памяти, выделенной для хранения дерева
void ReleaseTree(void);
// Функция вычисления максимума
__inline double max(double v1, double v2);
// Функция вычисления премии по значению в узле
__inline double func(double value);
// Вычисление средней премии – основная расчетная функция
double GetExpectation(void);
// Головная функция
int main(void);
```

Прокомментируем основные функции.

Функция **InputTree** предназначена для ввода исходных данных в соответствии с условием задачи. В этой функции вычисляется количество узлов и ветвей, выделяется память для хранения дерева, заполняются вспомогательные массивы **power** и **index**, инициализируется датчик случайных чисел и происходит заполнение дерева. При этом гарантируется выполнение условия $\sum_{j=0}^{B-1} P_{L,L+1}^{i,j} = 1$. Рекомендуем на этапе отладки отключать инициализацию датчика, для того чтобы при каждом запуске работа велась с одним и тем же деревом, или инициализировать дерево заранее заготовленными данными, для которых вручную подсчитан правильный ответ.

Функция **func** предназначена для обобщения задачи на случай, когда размер премии при открывании двери является некоторой функцией от *x*. В рассматриваемой сейчас постановке она просто возвращает значение *x*.

Функция **GetExpectation** рассчитывает средний размер премии по дереву при помощи описанного выше метода. Приведем ее реализацию:

```
// Вычисление среднего
// Алгоритм базируется на следующих соотношениях:
// E_l,i = СУММА по j=0__b-1 (Probability_l+1,j * E_l+1,j + value_l,i),
// где l - уровень, i - номер узла в уровне, l+1,j - узлы потомки узла l,i.
// Учитывая, что на последнем уровне E_l-1,i = Value_l-1,i
// Вычиляем рекуррентное соотношения из конца в начало.
// В итоге имеем сумму для нулевого узла - корня дерева
double GetExpectation(void)
{
  // Последний уровень
  int i, j, level;
  double sum;
  TreePart rp;
  for (j = 0; j < power[l-1]; j++)
    RobotTree[j + index[l-1]].expectation =
                                    func(RobotTree[j + index[l-1]].value);
  // Пересчет из конца в начало
  // Цикл по уровням
  for (level = l - 2; level >= 0; level--)
  {
    // Цикл по узлам уровня
    for (j = 0; j < power[level]; j++)
    {
      // Для узла level, j подсчитываем expectation
      // Цикл по потомкам
        sum  = func(RobotTree[ index[level] + j ].value);
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
    }
  }
  return RobotTree[0].expectation;
}
```

2.4.5. Параллельная реализация

Рассмотрим возможный вариант распараллеливания предложенной выше последовательной реализации. Акцент сделаем на корректности реализации, а не на ее производительности, которая не является целью данной лабораторной работы. Приведем лишь одно соображение по поводу производительности. Поскольку метод решения задачи предполагает, что последний уровень дерева обсчитывается отдельно от остальных, начнем наше распараллеливание именно с него. Если подумать, можно обнаружить первый «подводный камень» этой задачи, связанный не с корректностью, но с производительностью разрабатываемой реализации. Легко допустить неточность, посчитав, что распараллеливание на последнем уровне можно опустить. Казалось бы, в чем смысл отдельной работы

ради одного уровня? На самом деле смысл есть, поскольку последний уровень содержит наибольшее число узлов, и пренебрегать им при распараллеливании не стоит. Применим для распараллеливания обсчета директиву компилятора **`#pragma omp parallel for`**.

Соответствующий фрагмент кода будет выглядеть так:

```
  #pragma omp parallel for
    for (j = 0; j < power[l-1]; j++)
      RobotTree[j + index[l-1]].expectation =
                                    func(RobotTree[j + index[l-1]].value);
```

Заметим, что переменная **`j`** станет локализованной автоматически (согласно стандарту OpenMP), а все остальные по смыслу должны быть общими, поэтому задания дополнительных параметров директивы не требуется.

Перейдем к распараллеливанию основного блока кода, производящего обсчет дерева.

Естественный вариант состоит в разделении всех узлов каждого уровня между потоками. Этого можно добиться по крайней мере двумя способами. Первый состоит в размещении директивы **`#pragma omp parallel for`** перед циклом по узлам очередного уровня. В итоге получим:

```
    // Пересчет из конца в начало
    // Цикл по уровням
    for (level = l - 2; level >= 0;level--)
    {
    // Цикл по узлам уровня
    #pragma omp parallel for private(sum, i, rp)
      for (j = 0; j < power[level]; j++)
      {
        // Для узла level,j подсчитываем expectation
        sum  = func(RobotTree[ index[level] + j ].value);
        // Цикл по потомкам
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
      }
```

Проблема этого варианта состоит в том, что на каждой итерации внешнего цикла происходит «пробуждение» потоков в начале и «засыпание» в конце, что может плохо отразиться на производительности. Поэтому более правильным является вариант, в котором создание параллельной секции происходит один раз перед внешним циклом.

```
double GetExpectation(void)
{
  int i, j, level;
  double sum;
  TreePart rp;
  // Последний уровень
  #pragma omp parallel for
    for (j = 0; j < power[l-1]; j++)
      RobotTree[j + index[l-1]].expectation =
                                    func(RobotTree[j + index[l-1]].value);

  #pragma omp parallel
  {
    // Пересчет из конца в начало
    // Цикл по уровням
    for (level = l - 2; level >= 0; level--)
    {
    // Цикл по узлам уровня
    #pragma omp for private(sum, i, rp)
      for (j = 0; j < power[level]; j++)
      {
        // Для узла level,j подсчитываем expectation
```

```
        sum  = func(RobotTree[ index[level] + j ].value);
        // Цикл по потомкам
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
      }
    }
  }
  return RobotTree[0].expectation;
}
```

2.4.6. Анализ параллельной реализации

Собрав проект в соответствии с рекомендациями, изложенными в Описании ITC, запускаем инструмент отладки и обнаруживаем следующие диагностики:

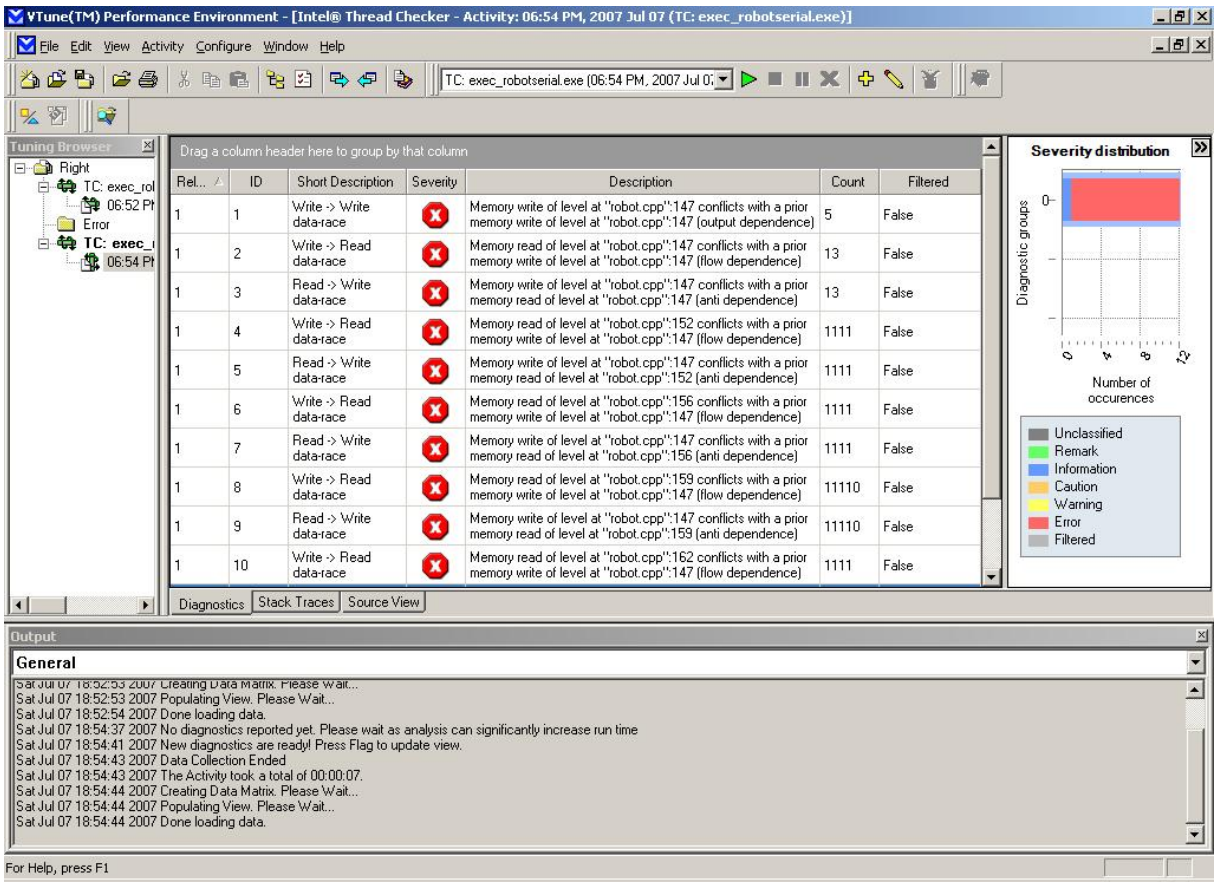


Рис. 19. Диагностика ITC в задаче о роботе

Развернув сообщения об ошибках, мы видим источник проблемы – гонки данных для переменной **`level`**. Действительно, предусмотрев локализацию при распараллеливании цикла, мы забыли об этом в директиве **`#pragma omp parallel`**. Исправим ошибку и приведем в заключение корректную реализацию.

```
double GetExpectation(void)
{
  int i, j, level;
  double sum;
  TreePart rp;
  // Последний уровень
  #pragma omp parallel for
    for (j = 0; j < power[l-1]; j++)
```

```
      RobotTree[j + index[l-1]].expectation =
                                    func(RobotTree[j + index[l-1]].value);

  #pragma omp parallel private(level)
  {
    // Пересчет из конца в начало
    // Цикл по уровням
    for (level = l - 2; level >= 0; level--)
    {
    // Цикл по узлам уровня
    #pragma omp for private(sum, i, rp)
      for (j = 0; j < power[level]; j++)
      {
        // Для узла level,j подсчитываем expectation
        sum  = func(RobotTree[ index[level] + j ].value);
        // Цикл по потомкам
        for (i = 0; i < b; i++)
        {
          rp = RobotTree[ index[level+1] + b * j + i ];
          sum = sum + rp.expectation * rp.probability;
        }
        RobotTree[ index[level] + j ].expectation = sum;
      }
    }
  }
  return RobotTree[0].expectation;
}
```

3. Дополнительные задания

1. Изучите стандартные примеры, поставляемые вместе с инструментом отладки Intel Thread Checker.

2. Изучите постановку задачи умножения матрицы на вектор, последовательные реализации различных алгоритмов, а также предлагаемые пути распараллеливания (см. документ mc_ppr07_forITC.doc). Проанализируйте прилагаемые параллельные реализации, содержащие ошибки (папка Code\MV). Выполните отладку прилагаемых программ, добейтесь их работоспособности.

3. Изучите постановку задачи умножения матрицы на матрицу, последовательные реализации различных алгоритмов, а также предлагаемые пути распараллеливания (см. документ mc_ppr08_forITC.doc). Проанализируйте прилагаемые параллельные реализации, содержащие ошибки (папка Code\MM). Выполните отладку прилагаемых программ, добейтесь их работоспособности.

4. Подумайте над задачей об обедающих философах. Рассмотрите другие варианты ее решения. Реализуйте их. Выполните отладку разработанных программ, добейтесь их работоспособности. В качестве одного из средств контроля используйте ITC.

4. Литература

4.1. Использованные источники информации

1. Intel® Thread Checker for Windows*. Getting Started Guide. Version 3.0. — Intel Corporation, 2006.
2. Intel® Thread Checker Help. Version 3.0. — Intel Corporation, 2006.
3. Intel® Thread Checker. Guide to Sample Code. Version 3.0. — Intel Corporation, 2006.

4.2. Рекомендуемая литература

4. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming.. – Reading, MA: Addison-Wesley (русский перевод Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).
5. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
7. Гергель В.П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
8. Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.:БХВ-Петербург, 2002.

4.3. Дополнительная литература

9. Березин И.С., Жидков И.П. Методы вычислений. – М.: Наука, 1966.

4.4. Информационные ресурсы сети Интернет

10. Сайт Лаборатории Параллельных информационных технологий НИВЦ МГУ – http://www.parallel.ru.
11. Официальный сайт OpenMP – www.openmp.org.
12. Официальный форум MPI – www.mpi-forum.org.