Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Инструменты параллельного программирования для систем с общей памятью

Лабораторная работа: Отладка параллельной программы с использованием Intel Thread Checker

Сысоев А.В., Мееров И.Б.
Кафедра математического обеспечения ЭВМ

Содержание

- Введение
- Отладка программ на примерах
- Задания для самостоятельной работы
- Литература

Введение…

- **Обычное программирование**: «*Любая программа содержит хотя бы одну ошибку*».
- **Параллельному программирование**: «*Параллельное программирование начинается с параллельных ошибок*».

Введение…

- Процесс выявления ошибок – **тестирование**.
- Процесс устранения ошибок – **отладка**.

- Отладка последовательных программ сложна сама по себе.
- Отладка параллельных программ еще сложнее.

- **Вопрос**: *Почему?*

Введение…

- Отладка последовательных программ сложна сама по себе.
- Отладка параллельных программ еще сложнее.

- **Вопрос**: *Почему?*
- **Ответ**: *Логика работы у параллельных программ существенно сложнее, чем у последовательных*.

- **Вопрос**: *Как бороться со сложностью отладки параллельных программ?*

Введение…

- **Вопрос**: *Как бороться со сложностью отладки параллельных программ?*
- **Ответ**:
  - *досконально изучить используемые средства распараллеливания (MPI, OpenMP, потоки…) и принципы их функционирования;*
  - *использовать инструментальную поддержку.*
- **Освоение средств ПП** – предмет изучения в базовых курсах по параллельному программированию (освоено).
- **Инструментальная поддержка** – тема данной лабораторной работы.

Введение

- В ЛР рассматривается один из инструментов отладки – Intel Thread Checker,
  - основан на сборе и автоматическом анализе информации по результатам выполнения программ;
  - предназначен для многопоточных и OpenMP программ.
- В качестве полигона для демонстрации ошибок и возможностей инструментов по их поиску и анализу используются:
  - *задача скалярного умножения векторов*;
  - *задача Дирихле для уравнения Пуассона*, решаемая методом Гаусса-Зейделя;
  - *задача об обедающих философах*;
  - *задача «о роботе»* - дополнительное задание*.

План работы

- Разбор типовых задач:
  - *задачи скалярного умножения векторов и матрично-векторного умножения*;
  - *задача Дирихле для уравнения Пуассона*, решаемая методом Гаусса-Зейделя;
  - *задача об обедающих философах*;
  - *задача «о роботе» *.*
- Для каждой из задач рассматриваются:
  - постановка задачи;
  - модель, метод решения, последовательная реализация;
  - варианты параллельных реализаций и их анализ.
- На типовых задачах иллюстрируются типовые ошибки и методы их обнаружения при помощи ITC, а также приемы их устранения.
- В конце дается задание для самостоятельной работы.

Отладка программ на примерах

- Задача о скалярном произведении векторов.
- Задача Дирихле.
- Задача об обедающих философах.
- Задача о роботе*.

Задача о скалярном произведении векторов

Задача о скалярном произведении векторов. Постановка задачи

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

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

Задача о скалярном произведении векторов. Последовательная реализация

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

Задача о скалярном произведении векторов. 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.

Задача о скалярном произведении векторов. Параллельная реализация 1

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

Задача о скалярном произведении векторов. Анализ параллельной реализации 1…

Запустим ITC:

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

Задача о скалярном произведении векторов. Анализ параллельной реализации 1

- 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.

Задача о скалярном произведении векторов. Параллельная реализация 1. Коррекция ошибок

- Реализовать эту схему в OpenMP можно различными способами, рассмотрим наиболее быстрый вариант, использующий параметр **reduction** директивы **parallel for**.

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

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

Задача о скалярном произведении векторов. Параллельная реализация 2

- Рассмотрим второй вариант параллельной реализации.
- Идея его несколько искусственна, но ее реализация позволит нам увидеть характерные ошибки при создании параллельных программ.
- Параллельная реализация 2.

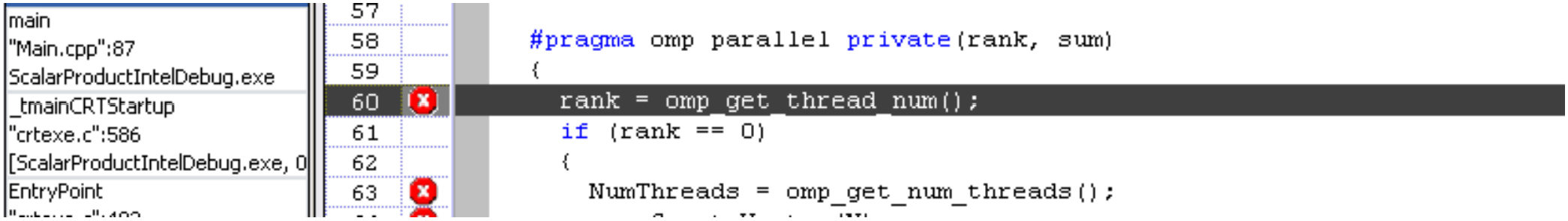
- Запустим реализацию на выполнение: результаты более чем странные. Программа работает некорректно, воспользуемся ITC.

Задача о скалярном произведении векторов. Параллельная реализация 2. Анализ…

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

Задача о скалярном произведении векторов. Параллельная реализация 2. Анализ…

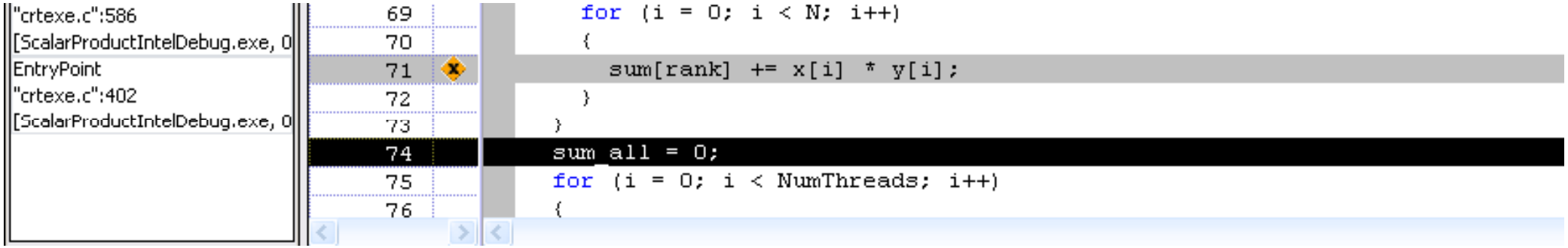
1	1	OpenMP - use of unsupported...		OpenMP - use of unsupported API at "ompScalar.cpp":60	1



- ITC «не понимает» run-time функции OpenMP (omp_get_num_threads()). Отсюда и большая часть остальных ошибок, которые вызваны тем, что значение переменной rank не известно ITC, и он находит конфликты там, где их на самом деле нет.
- Выходы:
  - отказаться от использования run-time функций OpenMP;
  - отказаться от компиляторного инструментирования, то есть удалить из опций проекта ключ /Qtcheck.

Задача о скалярном произведении векторов. Параллельная реализация 2. Анализ…

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



- Ошибка: переменная **sum** сделана **private**, что неверно. Этот массив специально был создан для подсчета потоками локальных сумм и должен быть общим, чтобы по окончании параллельной секции из него в переменную **sum_all** можно было собрать итоговую сумму.
- Исправление:

```
#pragma omp parallel private(rank/*, sum*/)
{
 ...
}
```

Задача о скалярном произведении векторов. Параллельная реализация 2. Анализ…

- Собираем приложение без ключа /Qtcheck.
- Запускаем 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

Задача о скалярном произведении векторов. Параллельная реализация 2. Анализ

- Суть ошибки: потоки, отличные от нулевого, успевают воспользоваться памятью до того, как она была выделена нулевым потоком.
- Решение: установить барьер.

```
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
…
```

Задача Дирихле

Задача Дирихле. Постановка задачи

- Рассмотрим задачу из области численного решения дифференциальных уравнений в частных производных, а именно задачу Дирихле для уравнения Пуассона.
- Постановка задачи:

  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 $g(x, y)$.

Задача Дирихле. Метод решения

- Используя распространенный метод конечных разностей (он же метод сеток) перепишем уравнение Пуассона в конечно-разностной форме:

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

- Разрешив его относительно u_{ij}, получим

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

- 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.

Задача Дирихле. Последовательная реализация

```
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); //dmax используется для останова по точности
```

Задача Дирихле. Параллельная реализация 1

```
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); //dmax используется для останова по точности
```

Задача Дирихле. Анализ параллельной реализации 1…

- Проанализируем код на наличие ошибок.
- Результаты последовательной и параллельной версии не совпадают. Однако, это не является критерием наличия ошибки.
- Причина: схема вычислений изменилась.
- Несколько запусков параллельной версии, выполненных друг за другом, показывают, что результаты параллельной версии различаются и от запуска к запуску.

  В данном случае и это не является критерием наличия ошибки.
- Причина: условия запуска влияют на порядок вычислений.

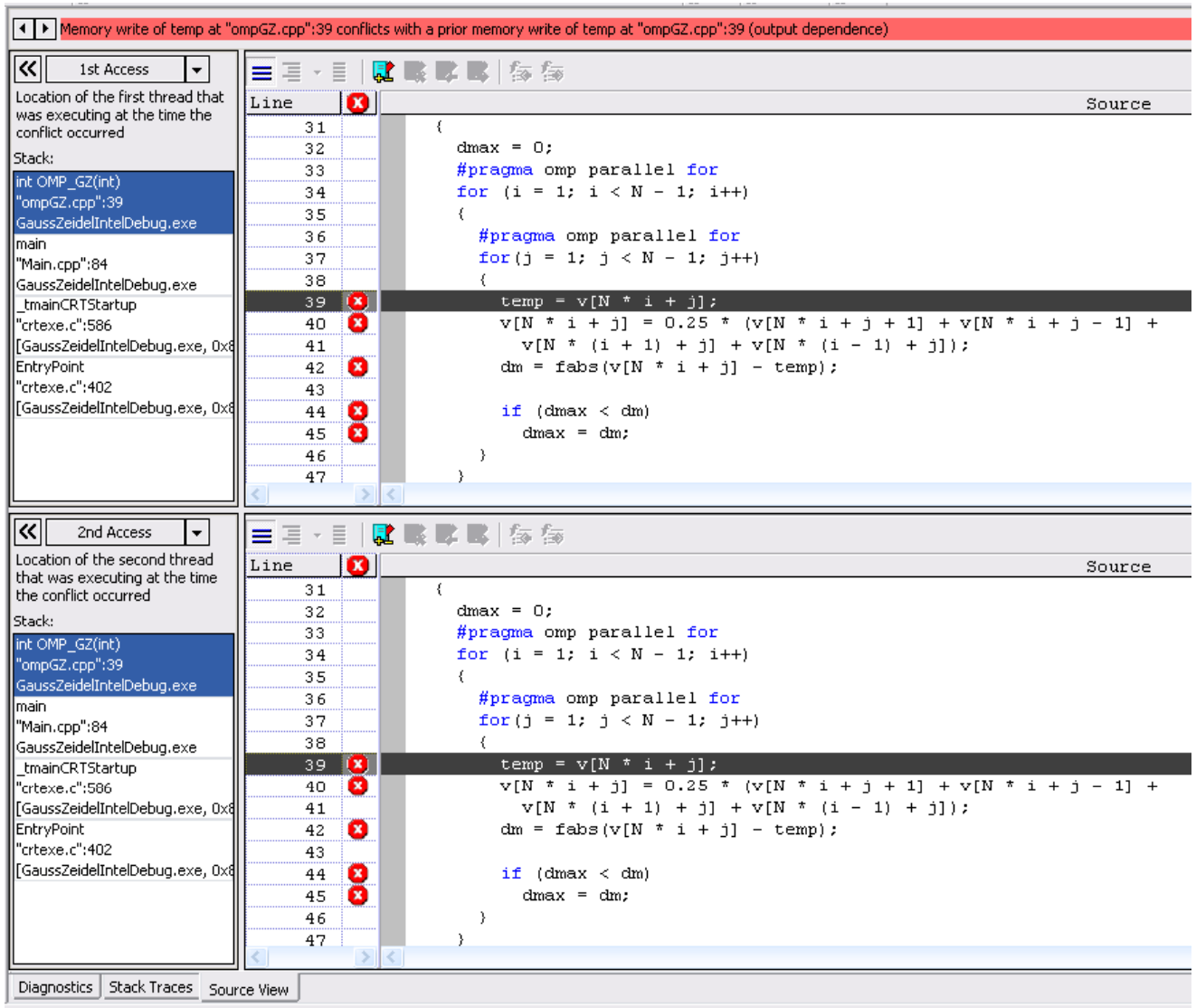
Задача Дирихле. Анализ параллельной реализации 1…

- Отладка и тестирование параллельных программ – сложная задача.
- **Подход к решению**: сравнение результатов последовательной версии и параллельной версии в смысле заданной точности.
- **Дополнительный контроль**: использование инструментов отладки.
- Проанализируем код с помощью Intel Thread Checker.

Задача Дирихле.
Анализ параллельной реализации 1. Диагностика ITC

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

Задача Дирихле.
Анализ параллельной реализации 1. Гонка данных



Задача Дирихле. Параллельная реализация 1. Коррекция ошибки…

- Диагностика ITC указывает нам 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**.

Задача Дирихле. Параллельная реализация 1. Коррекция ошибок

- Параллельная реализация 1.

Задача Дирихле. Параллельная реализация 1. После коррекции

- Тем не менее, ITC показывает наличие ошибки. Прав ли ITC?

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

Задача Дирихле. Параллельная реализация 2

```
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);
```

Задача Дирихле. Анализ параллельной реализации 2…

- В связи с изменившейся схемой работы, как и рассмотренная ранее параллельная версия, данная также может не давать идентичные результаты по сравнению с последовательным кодом, что, конечно, является дополнительным усложняющим моментом для «ручной» отладки.
- Как и в предыдущем случае, из нескольких запусков подряд видно, что результаты параллельной версии существенно меняются от запуска к запуску.
- Проанализируем код с помощью Intel Thread Checker.

Задача Дирихле.
Анализ параллельной реализации 2…

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	20273
1	2	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	3	**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	4	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	5	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	6	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	7	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	8	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

Задача Дирихле. Анализ 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**.

Задача Дирихле. Параллельная реализация 2. Коррекция ошибок

- Параллельная реализация 2.

Задача Дирихле. Параллельная реализация 2. После коррекции

- Тем не менее, ITC показывает наличие ошибки. Прав ли ITC?

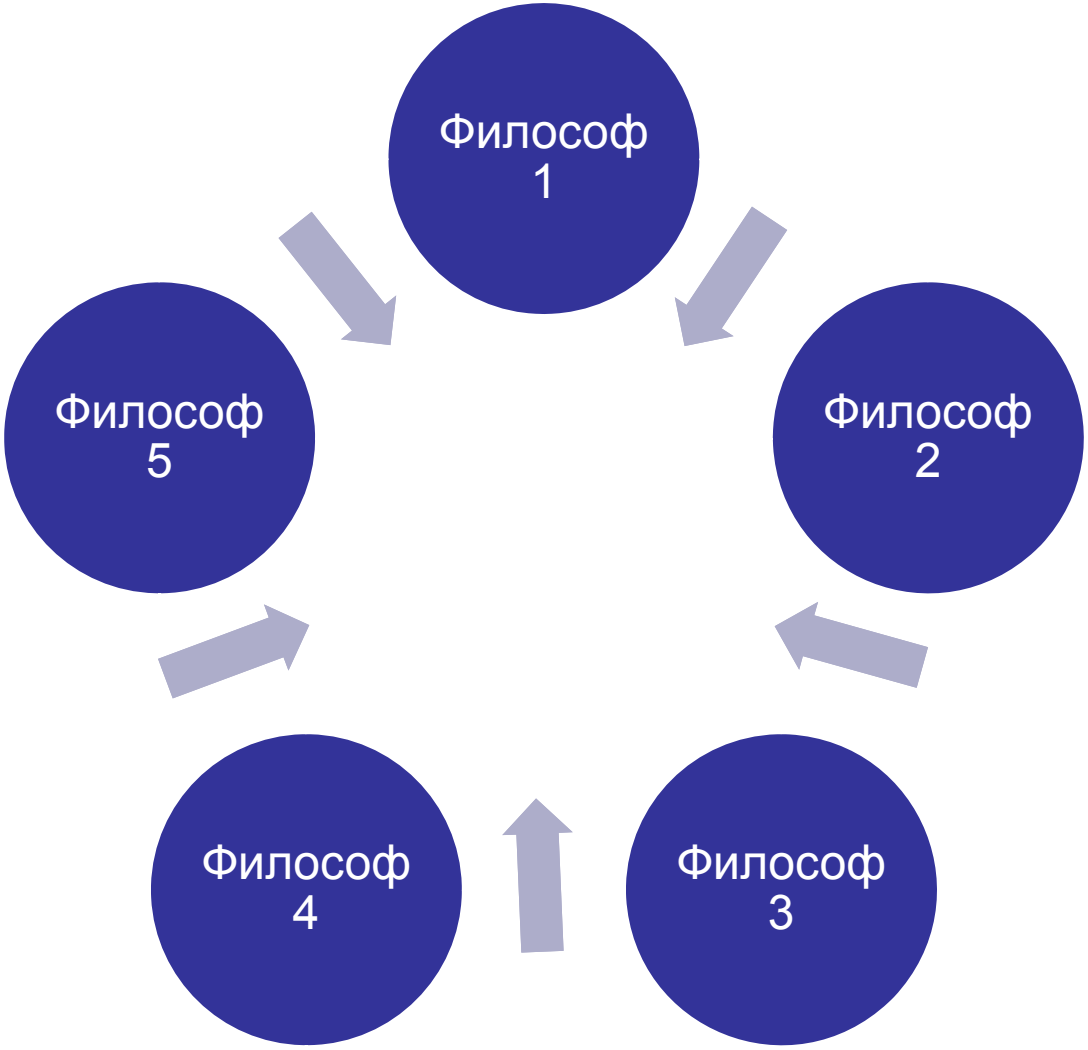
Re...	ID	Short Description	Severity	Description	Count
1	**1**	**Write -> Read data-race**		**Memory read of v[] at "ompGZ.cpp":233 conflicts with a prior memory write of v[] at...**	**198675**
1	2	Read -> Write data-race		Memory write of v[] at "ompGZ.cpp":233 conflicts with a prior memory read of v[] 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

Задача
об обедающих философах

Задача об обедающих философах. Цель рассмотрения и постановка задачи

- На примере данной классической задачи мы продемонстрируем еще одну типичную ошибку – **тупик** (**deadlock**), ее диагностику при помощи ITC и один из способов устранения.
- Рассмотрим **задачу Дейкстры** в вольной формулировке:
  - 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.
- **Задача**: Реализовать симулятор, демонстрирующий заседание философов.

Задача об обедающих философах. Графическая иллюстрация



Задача об обедающих философах. Параллельная реализация, вариант 1. Идея

- Реализуем требуемый симулятор на основе потоков Windows Threads.
- Идея реализации:
  - 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.

Задача об обедающих философах. Параллельная реализация, вариант 1. Объявления

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

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

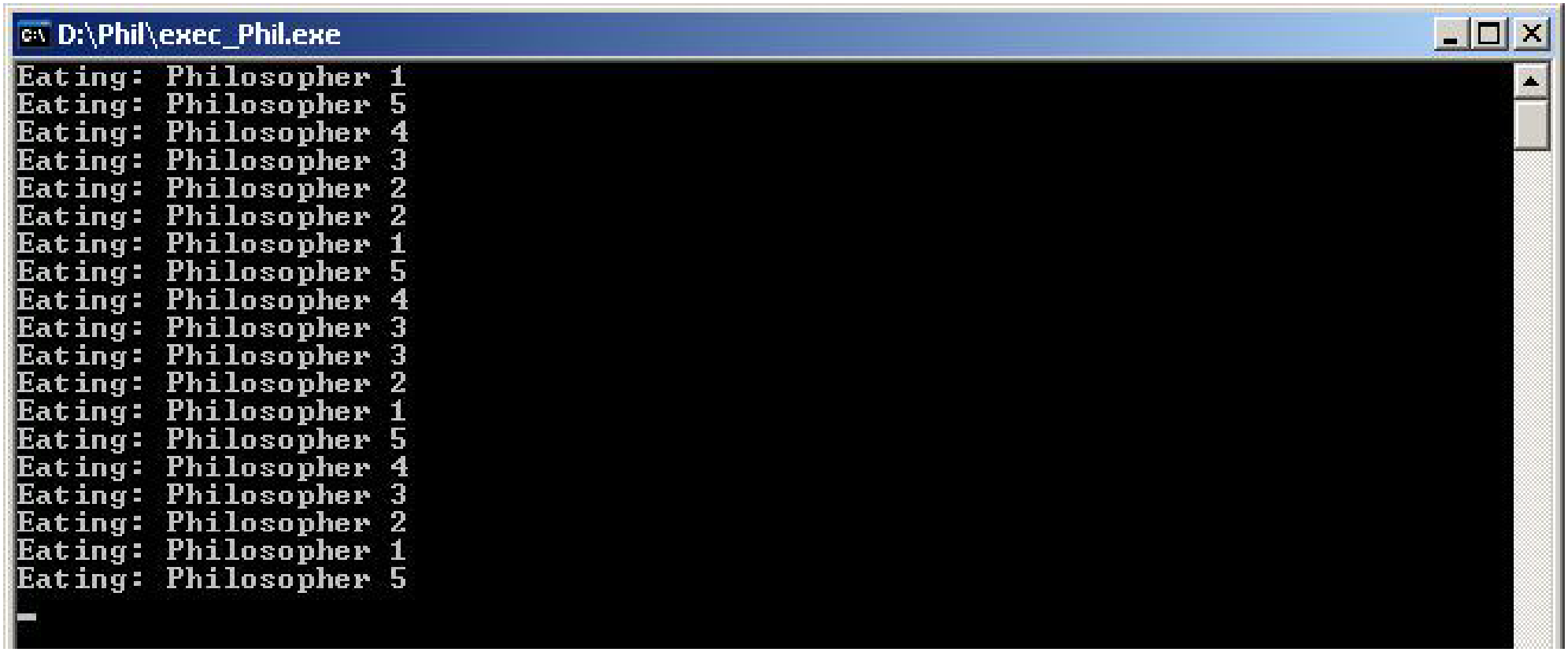
Задача об обедающих философах. Параллельная реализация, вариант 1. Функция потока

```
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);
}
```

Задача об обедающих философах. Параллельная реализация, вариант 1. Функция main

```
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);
}
```

Задача об обедающих философах. Параллельная реализация, вариант 1. Результаты



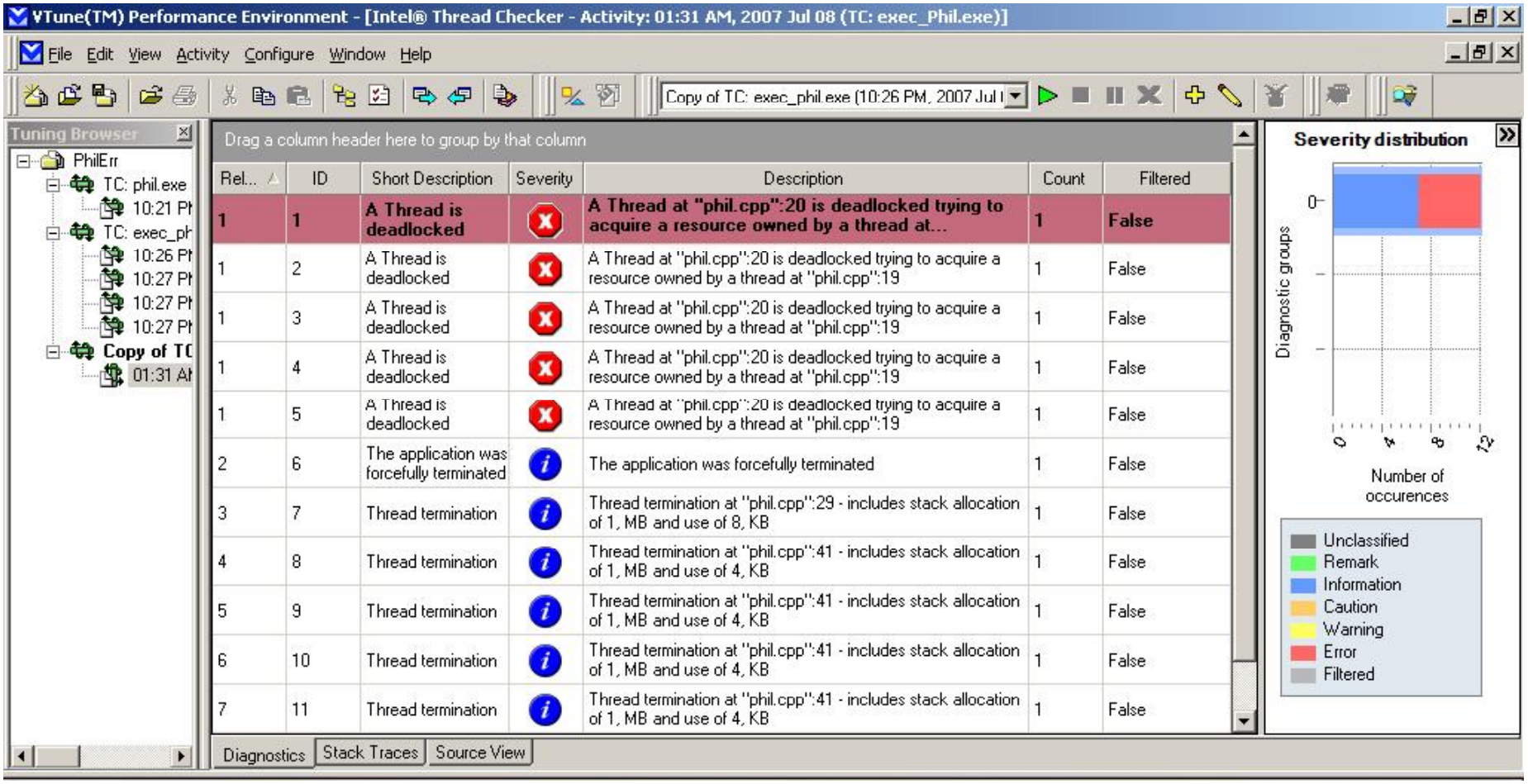
Результаты запуска параллельной реализации 1 (один из запусков)

Задача об обедающих философах. Параллельная реализация, вариант 1. Анализ…

- Результаты варьируются от запуска к запуску.
- В некоторый момент приложение перестает выводить на консоль информацию – зависает.

- Попробуем понять, в чем причина: запускаем ITC.

Задача об обедающих философах. Параллельная реализация, вариант 1. Анализ…



Задача об обедающих философах. Параллельная реализация, вариант 1. Анализ

- ITC показал 5 сообщений об ошибке – по одному на каждый из потоков. Разновидность ошибки – тупик (deadlock).
- **Источник ошибки**: логика работы программы допускает ситуации, в которых 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. В результате возникает тупик.

Задача об обедающих философах. Параллельная реализация, вариант 2. Идея

- **Цель модификаций**: исключить тупики.
- Источник тупиков не столько в некорректности реализации, сколько в самой постановке задачи.

- **Один из возможных способов решения проблемы**: изменение модели поведения философа. 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.

Задача об обедающих философах.
Параллельная реализация, вариант 2. Модификация

```
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);
}
```

Задача об обедающих философах. Параллельная реализация, вариант 2. Анализ

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

Задача о роботе*

Задача о роботе. Постановка задачи…

- В гипотетической лаборатории искусственного интеллекта 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).

Задача о роботе. Постановка задачи

- Считая известным:
  - количество дверей (*B*),
  - количество уровней полигона (*L*),
  - количество монет, лежащее за дверью *i* на уровне *L* (x_L^i),
  - 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:

$$B \le 20, L \le 10.$$

Задача о роботе. Модель…

Исходные данные:

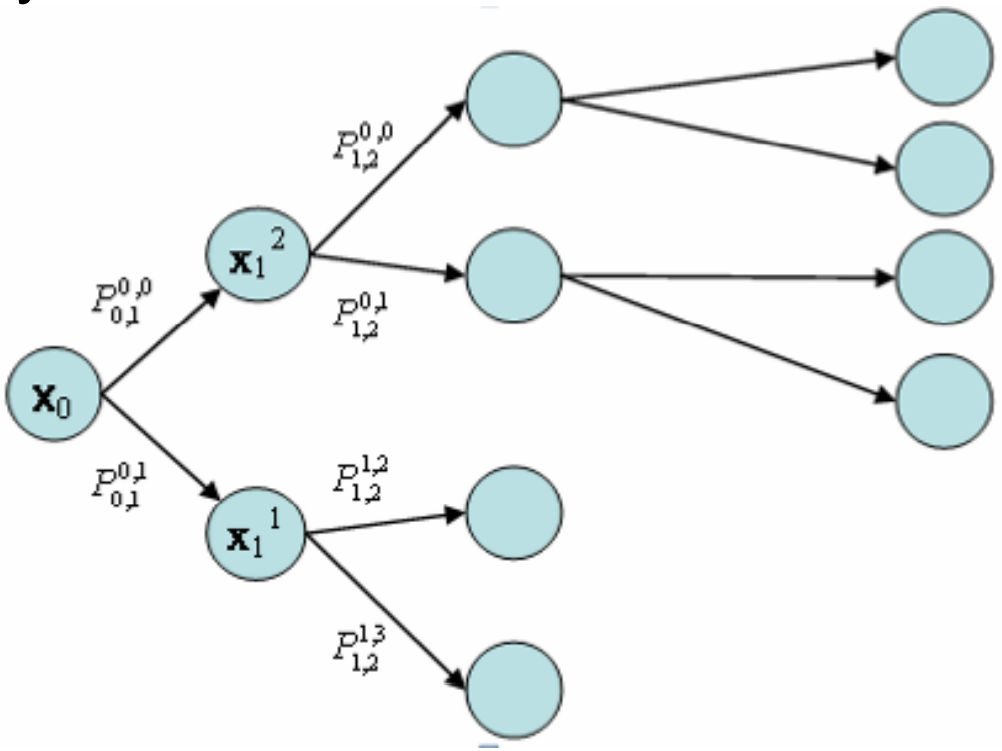
- *B* – количество вариантов пути, выбираемых роботом на каждом шаге.
- *L* – количество уровней полигона.
- $P_{L,L+1}^{i,j}$ – вероятность попадания из узла *i* уровня *L* в узел *j* уровня *L+1*.
- x_L^i – количество монет, лежащее за дверью *i* уровня *L*.

Ограничения:

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

Задача о роботе. Модель

- **Вопрос**:
  - Модель представления полигона?
- **Ответ**:
  - Дерево степени *B* и глубиной *L*.



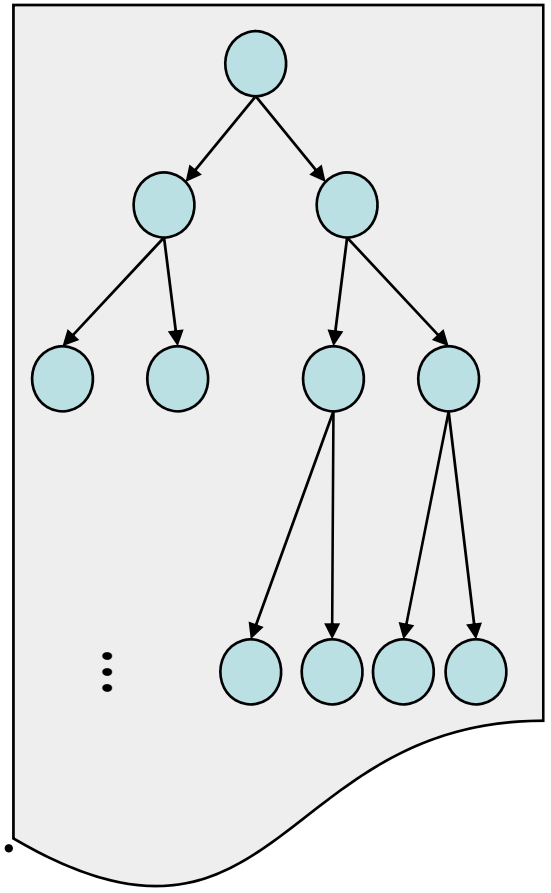
Задача о роботе. Метод решения…

- Вопрос:
  - Как сосчитать среднюю премию?
- Ответ:
  - Пусть E_d^i – средняя премия, которая может быть получена, если робот начинает путь в узле *i* уровня *d*.
  - Тогда

$$E_d^i = x_d^i + \sum_{j=0}^{B-1} P_{d,d+1}^{i,j} \times E_{d+1}^j ,$$

где $d = \overline{0; l-2};\ i = \overline{0; B-1}$ $(i = 0$ при $d = 0)$.

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

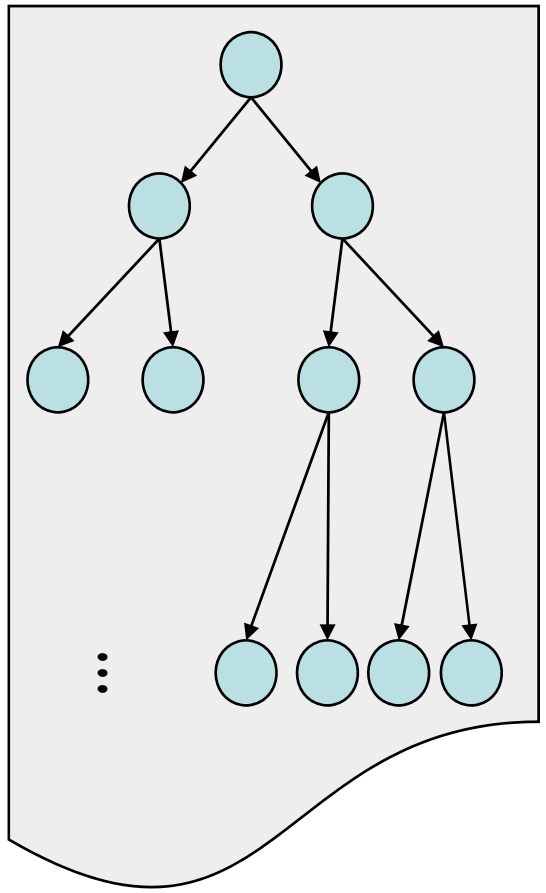


Задача о роботе. Метод решения

$$E_d^i = x_d^i + \sum_{j=0}^{B-1} P_{d,d+1}^{i,j} \times E_{d+1}^j \ ,$$

где $d = \overline{0; l-2}$; $i = \overline{0; B-1}$ ($i = 0$ при $d = 0$).

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



Очевидный метод состоит в вычислении значений *E* на последнем уровне дерева с последующим пересчетом из конца в начало по первой рекуррентной формуле.

Задача о роботе. Последовательная реализация. Проектирование структур данных…

```
const int b = 10;           // Степень дерева
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;
```

Задача о роботе. Последовательная реализация. Проектирование структур данных…

```
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.
```

Задача о роботе. Последовательная реализация. Проектирование структур данных

```
// 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];
```

Задача о роботе. Последовательная реализация. 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);
```

Задача о роботе. Последовательная реализация. Основная расчетная функция

Реализация функции

Задача о роботе. Параллельная реализация. Идея

- Будем использовать OpenMP.
- Последний уровень дерева обсчитывается отдельно – не забудем распараллелить этот расчет, так как в нем задействовано наибольшее количество узлов.
- В основном расчетном блоке будем распараллеливать внутренний цикл. Вынесем создание параллельной области из тела внешнего цикла, чтобы избежать пробуждения/засыпания потоков на каждой итерации.

Параллельная реализация, версия 1

Задача о роботе. Параллельная реализация. Результаты

- Запустив приложение, обнаруживаем, что результаты вычислений
  - отличаются от запуска к запуску;
  - **существенно** не совпадают с «эталонной» последовательной версией.

- Соберем проект с установками для ITC, запустим ITC.

Задача о роботе. Параллельная реализация. Анализ…

Drag a column header here to group by that column

Rel...	ID	Short Description	Severity	Description	Count	Filtered
1	1	Write -> Write data-race		Memory write of level at ''robot.cpp'':147 conflicts with a prior memory write of level at ''robot.cpp'':147 (output dependence)	5	False
1	2	Write -> Read data-race		Memory read of level at ''robot.cpp'':147 conflicts with a prior memory write of level at ''robot.cpp'':147 (flow dependence)	13	False
1	3	Read -> Write data-race		Memory write of level at ''robot.cpp'':147 conflicts with a prior memory read of level at ''robot.cpp'':147 (anti dependence)	13	False
1	4	Write -> Read data-race		Memory read of level at ''robot.cpp'':152 conflicts with a prior memory write of level at ''robot.cpp'':147 (flow dependence)	1111	False
1	5	Read -> Write data-race		Memory write of level at ''robot.cpp'':147 conflicts with a prior memory read of level at ''robot.cpp'':152 (anti dependence)	1111	False
1	6	Write -> Read data-race		Memory read of level at ''robot.cpp'':156 conflicts with a prior memory write of level at ''robot.cpp'':147 (flow dependence)	1111	False
1	7	Read -> Write data-race		Memory write of level at ''robot.cpp'':147 conflicts with a prior memory read of level at ''robot.cpp'':156 (anti dependence)	1111	False
1	8	Write -> Read data-race		Memory read of level at ''robot.cpp'':159 conflicts with a prior memory write of level at ''robot.cpp'':147 (flow dependence)	11110	False
1	9	Read -> Write data-race		Memory write of level at ''robot.cpp'':147 conflicts with a prior memory read of level at ''robot.cpp'':159 (anti dependence)	11110	False
1	10	Write -> Read data-race		Memory read of level at ''robot.cpp'':162 conflicts with a prior memory write of level at ''robot.cpp'':147 (flow dependence)	1111	False

Diagnostics | Stack Traces | Source View

Задача о роботе. Параллельная реализация. Анализ

- Обнаружены гонки данных для переменной level.
- Действительно, предусмотрев локализацию при распараллеливании цикла, мы забыли об этом в директиве **#pragma omp parallel**.
- Исправим ошибку и получим корректную реализацию.

Параллельная реализация, версия 2

Задания для самостоятельной работы

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

Использованные источники информации

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

Рекомендуемая литература

- 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).
- Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
- Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
- Гергель В.П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
- Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.:БХВ-Петербург, 2002. – 400 с.:илл.

Дополнительная литература

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

Информационные ресурсы сети Интернет:

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

Авторский коллектив

- Сысоев Александр Владимирович,
  ассистент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.
- Мееров Иосиф Борисович,
  к.т.н., доцент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.