Летняя школа

«Суперкомпьютерное моделирование и визуализация в научных исследованиях»

МГУ ВМК, 2010

Параллельное программирование с использованием модели вычислений Unified Parallel C

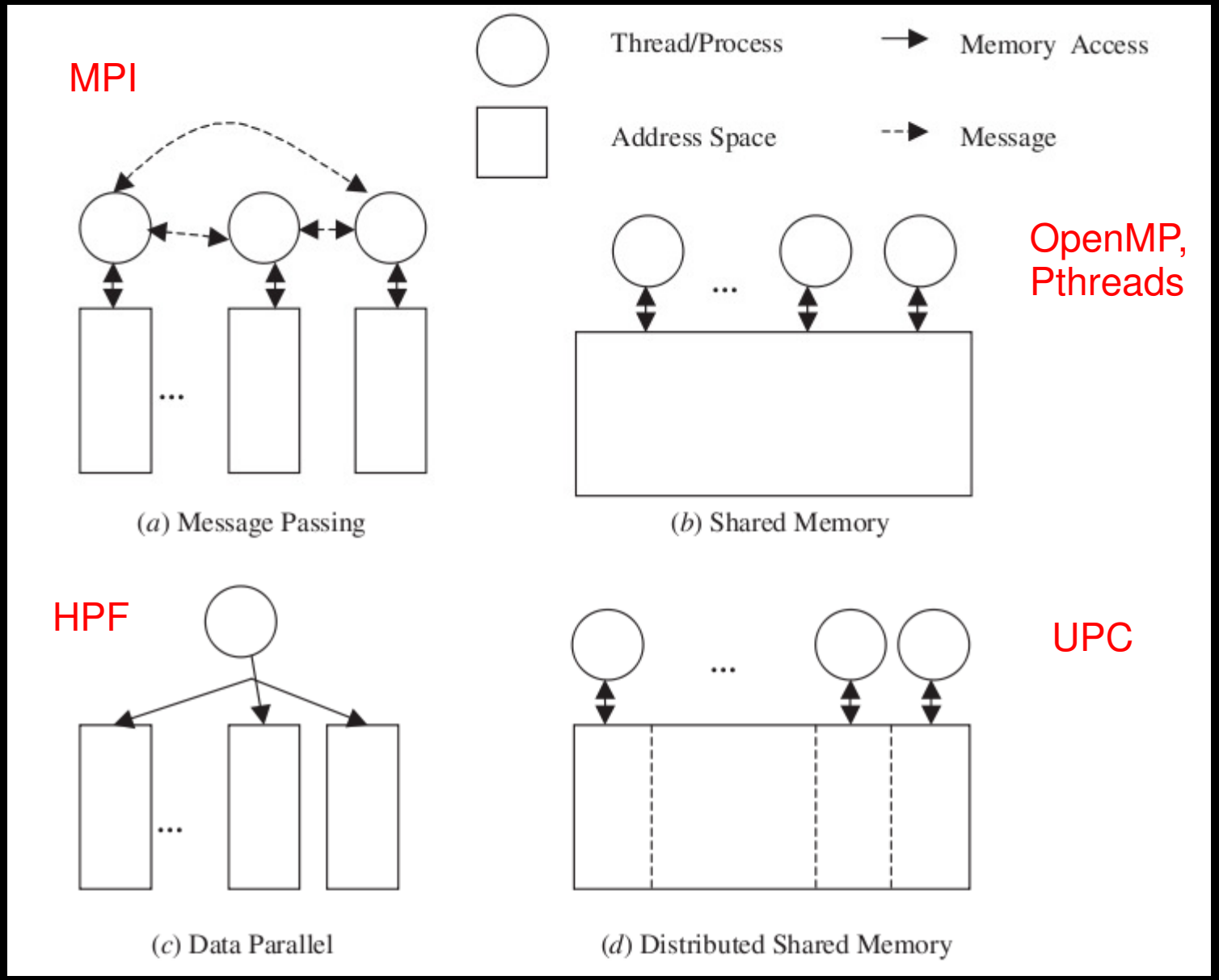
Федулова И.А.
инженер-программист, к.ф.-м.н.
IBM Russian Systems and Technology Laboratory
irina@ru.ibm.com

12.07.2010

На основе материалов “UPC Tutorial”, Tarek El-Ghazawi, PGAS09 и “UPC Tutorial”, Adam Leko, www.hcs.ufl.edu

План лекции

- Концепция Partitioned Global Address Space
- Стандарт UPC
  - Расширение языка C
  - Программа состоит из множества нитей (threads)
  - Общая память
- Berkeley UPC
  - Networking layer
  - Compiler suite
- Элементы языка UPC
  - Модель выполнения
  - Модель памяти
  - Shared, private variables
  - Распределение shared массивов по процессорам
  - Указатели и динамическое выделение памяти
  - Синхронизация
  - Распределение итераций циклов
- Примеры



"UPC: Distributed Shared Memory Programming", T.A.Ghazawi, W.Carlson, T.Sterling, K.Yelick, 2005

Концепция Partitioned Global Address Space

- Традиционная схема
  - Распределенная память => message passing модели (MPI, Charm++, …)
    - Хорошо масштабируется
    - Сложно программировать
  - Общая память => OpenMP, POSIX threads, …
    - Не очень хорошо масштабируется
    - Не так сложно программировать
  - Гибридное MPI + OpenMP программирование
    - Максимальная производительность
    - Максимальные усилия

«Ассемблер»

- Global Address Space
  - С точки зрения программиста память общая => сравнительная легкость программирования
  - Параллелизм по типу SPMD, как в MPI
  - Языки, поддерживающие концепцию PGAS
    - C => UPC
    - Fortran => Co-Array Fortran
    - Java => Titanium
    - Chapel (Cray)
    - X10 (IBM)

«Языки высокого уровня»

Unified Parallel C

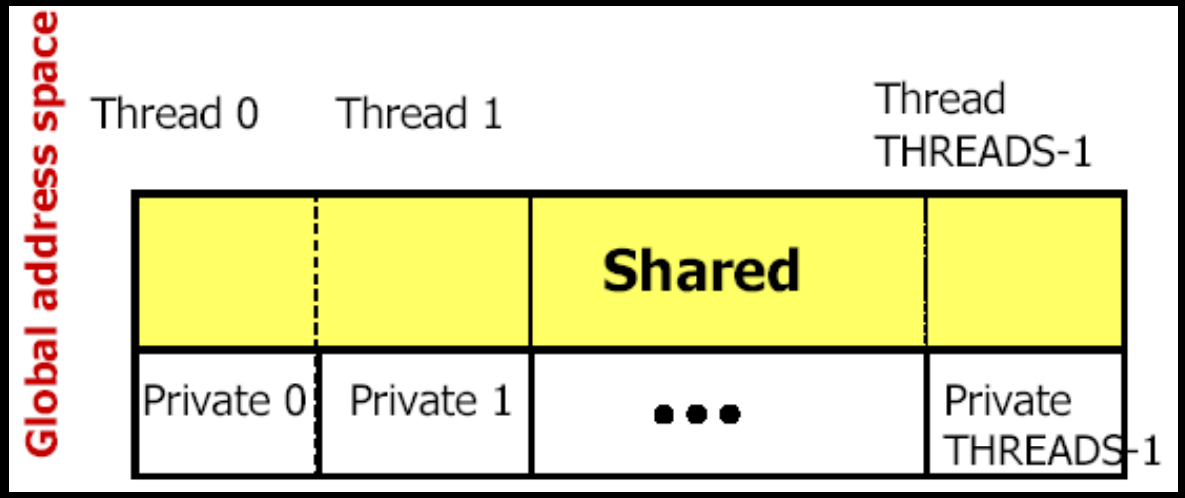
- Расширение стандарта ISO C 99
  - Явно параллельная модель выполнения (an explicitly parallel execution model)
  - Общее адресное пространство
  - Примитивы синхронизации и модели консистентности памяти
  - Примитивы для работы с памятью
- Философия языка похожа на C
  - Много возможностей
  - Требуется аккуратность
- Существуют различные реализации стандарта UPC
  - UPC @ GWU – разрабатывают стандарт http://upc.gwu.edu
  - Berkeley UPC
  - UPC @ MTU
  - UPC @ Florida
  - GCC UPC
- Поддерживается на различных HPC платформах
- Доступны средства разработки и отладки
  - TotalView
  - Eclipse Parallel Tools Platform

Berkeley UPC

- Berkeley UPC http://upc.lbl.gov
  - Философия
    - Переносимость
    - Высокая производительность
    - Large scale multiprocessors
  - Состав пакета
    - Легковесная среда выполнения и библиотека работы с физической сетью
      - GASNet
    - Компилятор
    - Benchmarks

Основные элементы Unified Parallel C

- Два типа переменных
  - private
    - по умолчанию
  - Shared
    - специальная директива

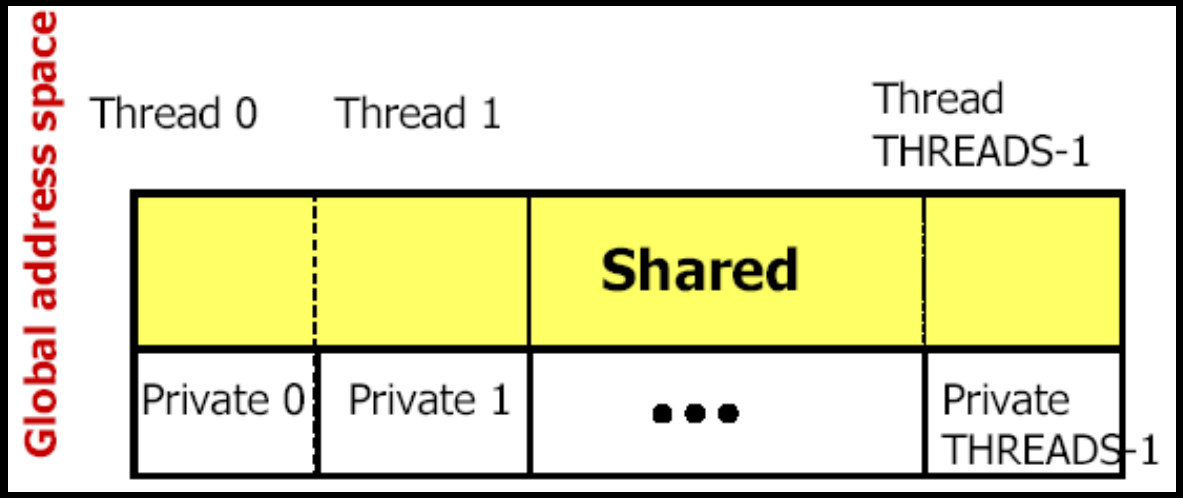


- Shared массивы и распределение данных по процессорам
- Распределение вычислений по процессорам
- Синхронизация и обеспечение консистентности памяти
  - Барьеры
  - Мьютексы (locks)

Модель выполнения в UPC

- Несколько потоков выполняют одну и ту же программу, но обрабатывают разные данные (SPMD)
  - Подобно MPI
  - Нет неявной синхронизации
  - MYTHREAD – номер текущего потока, THREADS – общее число потоков
  - Число потоков может задаваться
    - во время компиляции
    - во время выполнения (подобно mpirun –np N)
- Синхронизация применяется только тогда, когда это необходимо
  - Барьеры
  - Блокировки (Locks)
  - Контроль консистентности памяти

Модель памяти UPC



- Адресное пространство потока логически разделено на две части
  - Собственное пространство (private)
  - Разделяемая память (shared)
- Любой поток имеет доступ к разделяемой памяти
- Разделяемая память физически распределена между потоками
  - Программист имеет возможность узнать, где физически расположены данные, и учитывать это
- Доступны примитивы динамического выделения разделяемой памяти
- Доступны указатели на разделяемые области памяти

Пример 1. Сумма двух векторов

```
//vect_add.c

#define N 1000

int v1[N], v2[N], v1plusv2[N];

void main()
{
    int i;
    for (i=0; i<N; i++)
        v1plusv2[i] = v1[i] + v2[i];
}
```

Пример 1. Сумма двух векторов

```
//vect_add.upc

#include <upc.h>

#define N 1000

shared int v1[N], v2[N], v1plusv2[N];

void main()
{
    int i;
    upc_forall (i=0; i<N; i++; &v1plusv2[i])
        v1plusv2[i] = v1[i] + v2[i];
}
```

Разделяемые (shared) и локальные (private) переменные

- Скаляры

```
int a;          // своя переменная у каждого потока

shared int b; // одна разделяемая переменная для всех потоков
              // физически расположена на потоке 0
```

- Массивы

```
shared int x[THREADS];      // каждый элемент расположен на отдельном
                            // потоке, но весь массив доступен всем

shared int y[10][THREADS]; // по 10 элементов на потоке
```

- Разделяемые переменные не могут быть автоматическими
  - Должны быть либо глобальными
  - Либо объявлены статическими `static shared int c;`

Физическое распределение переменных

```
// THREADS == 3
Shared int x;
shared int y[THREADS];
int z;
```

MYTHREAD = 0	MYTHREAD = 1	MYTHREAD = 2
x		
y[0]	y[1]	y[2]
z	z	z

Распределение разделяемых массивов

- round-robin
  - По умолчанию
- блочное распределение

```
shared [BLOCK_SIZE] int c[N];
```

```
// THREADS = 4
shared [3] int c[16];
```

MYTHREAD = 0	MYTHREAD = 1	MYTHREAD = 2	MYTHREAD = 2
c[0],c[1],c[2] c[12],c[13],c[14]	c[3],c[4],c[5] c[15]	c[6],c[7],c[8]	c[9],c[10],c[11]

- Автоматический размер блока

```
// THREADS = 4
shared [*] int c[10];
```

MYTHREAD = 0	MYTHREAD = 1	MYTHREAD = 2	MYTHREAD = 2
c[0],c[1],c[2]	c[3],c[4],c[5]	c[6],c[7],c[8]	c[9],c[10]

Распределение разделяемых массивов

- Пример блочного распределения двумерного массива
  - Даже у многомерных массивов блоки одномерные

```
// THREADS = 4
shared [3] int x[4][THREADS];
```

MYTHREAD = 0	MYTHREAD = 1	MYTHREAD = 2	MYTHREAD = 2
x[0][0]	x[0][3]	x[1][2]	X[2][1]
x[0][1]	x[1][0]	x[1][3]	x[2][2]
x[0][2]	x[1][1]	x[2][0]	x[2][3]
x[3][0]	x[3][3]		
x[3][1]			
x[3][2]			

Распределение разделяемых массивов

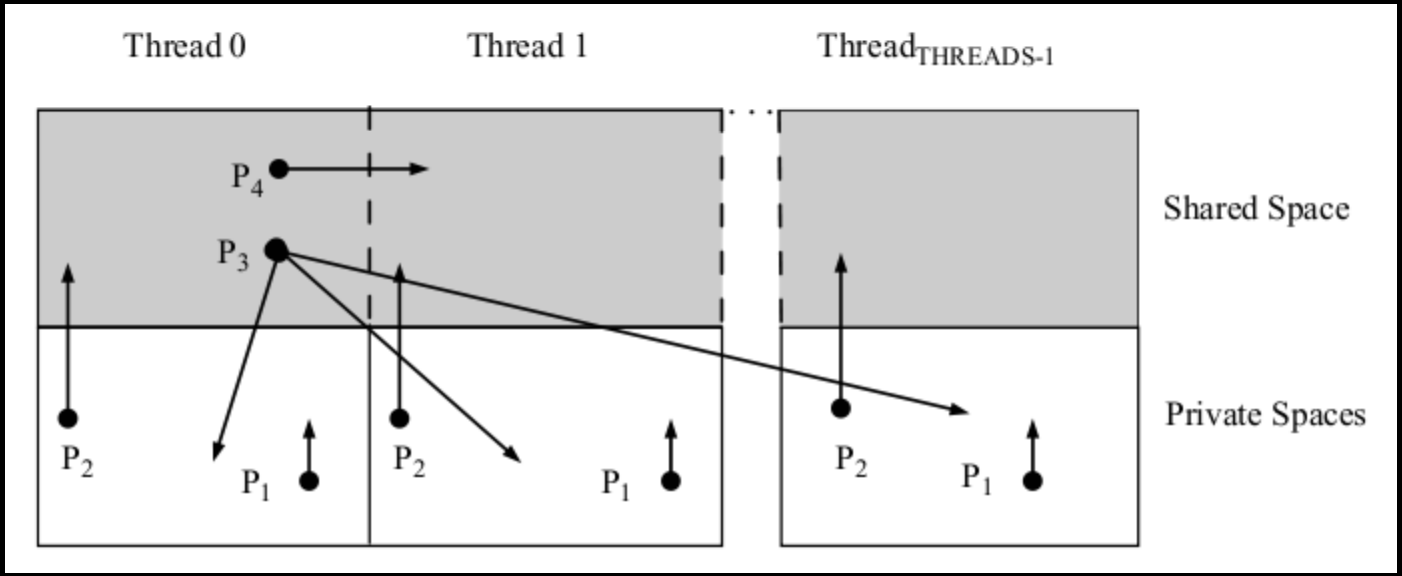
- Dynamic threads
  - число потоков THREADS задается во время запуска
- Static threads
  - число потоков THREADS задается на этапе компиляции

```
// допустимо и в static, и в dynamic
shared int x[10*THREADS];
shared [] int x[10];


// недопустимо в dynamic, т.к. невозможно определить,
// сколько элементов должно оказаться на каждом потоке,
// или полный размер массива
shared int x[10];
shared [] int x[THREADS];
Shared int x[10+THREADS];
```

Указатели

```
int *p1;                  // private to private
shared int *p2;           // private to shared
int *shared p3;           // shared to private [не имеет смысла!]
shared int *shared p4; // shared to shared
```



"UPC: Distributed Shared Memory Programming", T.A.Ghazawi, W.Carlson, T.Sterling, K.Yelick, 2005

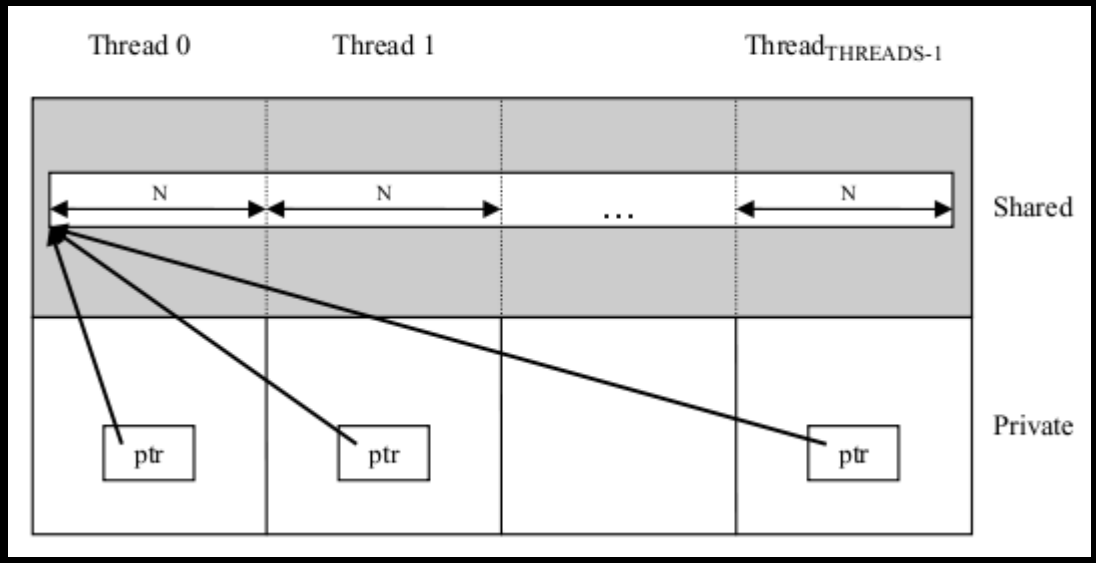
Динамическое выделение памяти

- В UPC возможно динамическое выделение памяти в разделяемом адресном пространстве
- `shared void *upc_global_alloc(size_t nblocks, size_t nbytes);`
  - Не коллективная функция – только один поток должен ее выполнять
  - Возвращает указатель на shared память
  - Вызывающий поток выделит «непрерывную область» памяти в разделяемом пространстве
  - Выделенная память будет распределена блочно, т.е. этот вызов эквивалентен вызову `shared [nbytes] char [nblocks * nbytes]`
- `shared void *upc_all_alloc(size_t nblocks, size_t nbytes)`
  - Коллективная функция – должна быть вызвана всеми потоками
  - Возвращает указатель на shared память
- `Void upc_free(shared void *);`
  - Освобождает выделенную память

Динамическое выделение памяти

```
shared [N] int *ptr;

ptr = (shared [N] int *) upc_all_alloc(THREADS, N*sizeof(int));
```

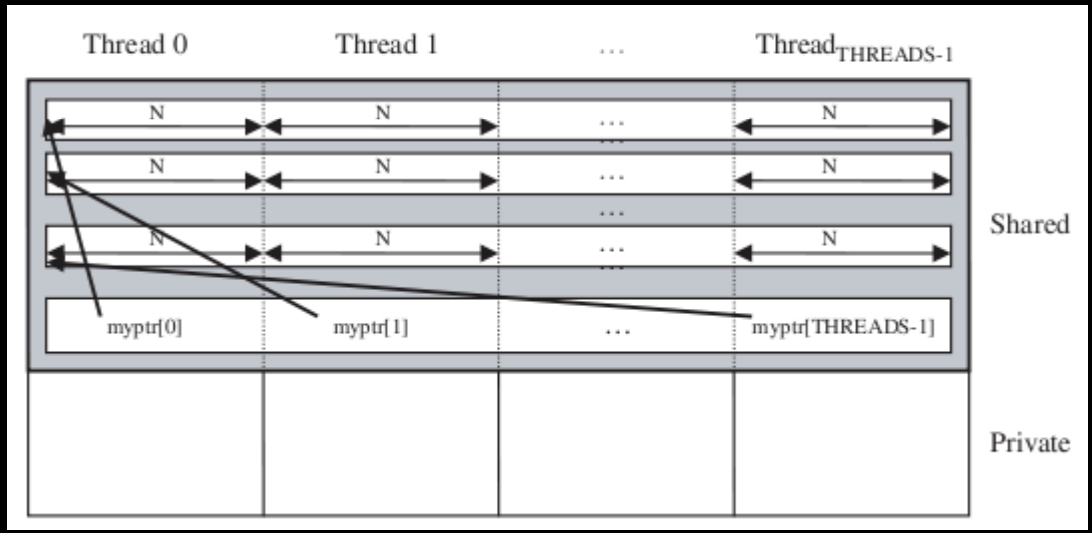


"UPC: Distributed Shared Memory Programming", T.A.Ghazawi, W.Carlson, T.Sterling, K.Yelick, 2005

Динамическое выделение памяти

```
shared [N] int *shared myptr [THREADS];

myptr [MYTHREAD] =
    (shared [N] int *) upc_global_alloc(THREADS, N*sizeof(int);
```



"UPC: Distributed Shared Memory Programming", T.A.Ghazawi, W.Carlson, T.Sterling, K.Yelick, 2005

Передача блоков памяти

- Для хорошей производительности необходимо минимизировать обращения к нелокальным областям памяти
- `upc_memget(void *dst, shared const void *src, size_t n);`
  - Копирование блока памяти из разделяемого пространства в локальное
- `upc_memput(void *dst, shared const void *src, size_t n);`
  - Копирование блока памяти из локального пространства в разделяемое
- `upc_memcpy(shared void *dst, shared const void *src, size_t n);`
  - Копирование блока памяти из разделяемого пространства в разделяемое

Распределение итераций циклов

- upc_forall
  - похож на оператор for
  - дополнительный аргумент – affinity
    - Целое число => будет выполнен потоком `MYTHREAD = affinity % THREADS`
    - Указатель на shared => будет выполнен потоком, на котором физически расположен элемент массива `MYTHREAD = upc_threadof(affinity)`
  - не гарантирует, что все потоки одновременно закончат выполнение

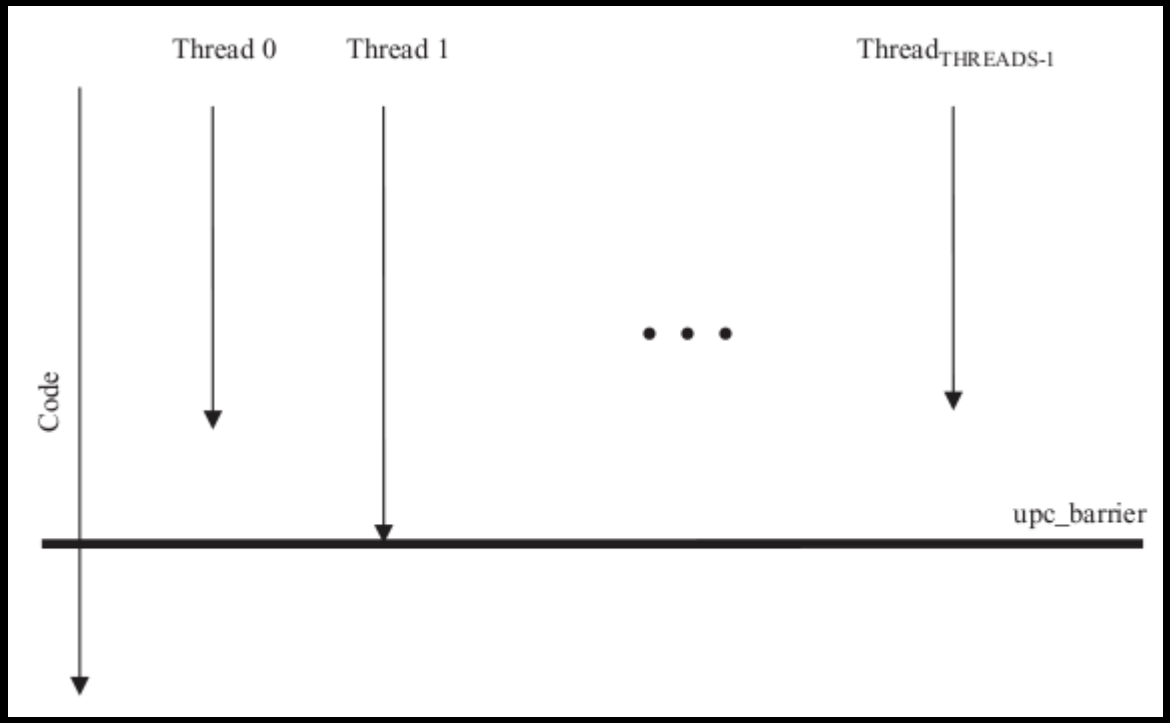
```
#include <upc.h>
#define TBL_SZ 12
main ()
{
  static shared int fahrenheit [TBL_SZ];
  static shared int step = 10;
  int celsius, i;
  upc_forall(i=0; i <TBL_SZ; i++; i) {
    celsius= step*i;
    fahrenheit[i]= celsius*(9.0/5.0) + 32;
  }

  upc_barrier;

  // print result
}
```

Синхронизация - барьер

```
upc_barrier;
```



"UPC: Distributed Shared Memory Programming", T.A.Ghazawi, W.Carlson, T.Sterling, K.Yelick, 2005

Синхронизация - мьютекс

- Мьютекс (в UPC – lock) – средство для контроля выполнения критических секций кода
- Создание и уничтожение

```
upc_lock_t *upc_all_lock_alloc(void);    // коллективная операция

upc_lock_t *upc_global_lock_alloc(void); // не коллективная

void upc_lock_free(upc_lock_t *ptr);
```

- Использование

```
void upc_lock(upc_lock_t *ptr);
void upc_unlock(upc_lock_t *ptr);
```

  - Если мьютекс закрыт, то поток, блокируется до тех пор, пока мьютекс не будет открыт потоком, закрывшим его

Синхронизация – мьютекс – пример

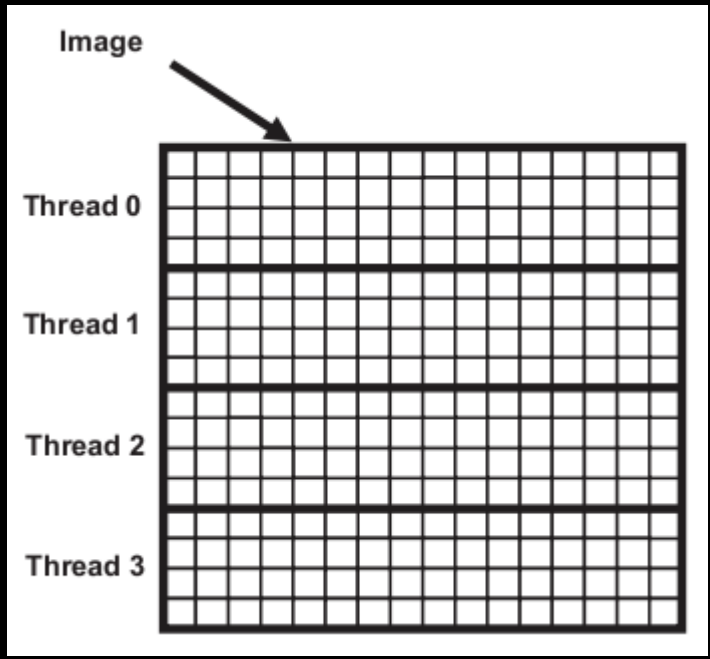
```
#include <upc.h>
#define N 1024

shared [N*N/THREADS] unsigned char img[N][N];
shared int hist[256];
upc_lock_t *lock;

void initialize(void) {
    lock = upc_all_lock_alloc();
    /* initialize the image img[][] */
}

int main(void) {
    int i, j;
    initialize();
    upc_barrier;
    upc_forall (i=0; i<N; i++; i*THREADS/N) {
        for (j=0; j<N; j++) {
            upc_lock(lock);
            hist[img[i][j]]++;
            upc_unlock(lock);
        }
    }
    upc_barrier;

    if (MYTHREAD == 0) print_hist();
    return 0;
}
```



"UPC: Distributed Shared Memory Programming", T.A.Ghazawi, W.Carlson, T.Sterling, K.Yelick, 2005

Пример 2. Перемножение матриц – ANSI C

```
#include <stdlib.h>

#define N 4
#define P 4
#define M 4

int a[N][P], c[N][M];
int b[P][M];

void main () {
  int i, j , l;

  for (i = 0 ; i < N; i++) {
    for (j = 0; j < M; j++) {
      c[i][j] = 0;
      for (l = 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b[l][j];
    }
  }
  return 0;
}
```

Пример 2. Перемножение матриц – MPI

```
/*
MPI parallel matrix multiplication using asynchronous message passing.
Each process will be assigned a number of rows.

**** slightly modified: rank 0 process responsible for first interval of
**** the matrix as well as the remainder.

The message passing calls used include synchronous as well asynchronous
send and receive, plus broadcast.

For simplicity, square matrices will be assumed and static, global arrays
used.
*/

#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>


// macro prints array AA, size rzXcz, using formated string fs.
// use as in PRINT2M(A,4,5,"%f ")
#define PRINT2M(AA,rz,cz,fs) { \
 int i, j; \
 for (i=0;i<rz;i++) \
   { for (j=0;j<cz;j++) printf(fs,AA[i][j]); \
     printf("\n"); } \
}


// size of NxN matrix:
#define N 500

// global static structures
double A[N][N];
double B[N][N];
double AB[N][N];   /* to hold result of A*B */

// conventional one-processor solution
void cmul()
{
  int i, j, k;
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      for(k=0,AB[i][j]=0; k<N; k++)
            AB[i][j] += A[i][k] + B[k][j];
} // cmul

void compinterval(int,int);  // prototype

int main(int argc, char** argv)
{
  int rank, size, interval, remainder, i, j;
  double time1, time2, time3;  // for timing measurements
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  MPI_Request ireq[128]; // , asynch request, assume size<128
  MPI_Status stat;       // status of asynch communication

  // compute interval size each process is responsible for,
  // rank 0 process will be responsible for the remainder
  interval = N/(size);
  remainder = N % (size);
```

```
if (rank==0)  // root/coordinator process
    {
      if (argc>1) srandom(atoi(*(argv+1)));  // random seed
      // generate random matrices
      for(i=0;i<N;i++) for(j=0;j<N;j++)
            { A[i][j] = random() % 4;
              B[i][j] = random() % 4;
            } // for

      time1 = MPI_Wtime();     // record time
      // For array B, we broadcast the whole array, however, the Bcast
      // operation is strange because it needs to be executed by all
      // processes.  There is no corresponding Recv for a Bcast
      MPI_Bcast(B,N*N,MPI_DOUBLE,0,MPI_COMM_WORLD); // send broadcast
      //      printf("%d: Bcast complete\n",rank);

      // Send intervals of array A to worker processes
      for(i=1;i<size;i++)
            MPI_Isend(A+(i*interval),interval*N,MPI_DOUBLE,i,i,
                        MPI_COMM_WORLD,ireq+i);
      for(i=1;i<size;i++)
            MPI_Waitany(size,ireq,&j,&stat); // join on sends

      compinterval(0,interval);                // local work
      compinterval(size*interval, remainder);  // remainder

      //get results from workers:
      for(i=1;i<size;i++)
            MPI_Irecv(AB+(i*interval),interval*N,MPI_DOUBLE,i,i,
                        MPI_COMM_WORLD,ireq+i);
      for(i=1;i<size;i++)
      {
            MPI_Waitany(size,ireq,&j,&stat);
            //          printf("received results from process %d\n",j);
      }

      time2 = MPI_Wtime();  // record time

      // run conventional algorithm:
      cmul();
      time3 = MPI_Wtime();

      printf("approx %d-process time Tp: %f sec.\n",size,time2-time1);
      printf("approx 1-process (conventional) time T1: %f sec.\n",time3-time2);
      printf("efficiency : %f\n",(time3-time2)/((time2-time1)*size));
      /* print :
      PRINT2M(A,N,N,"%.0f ");  printf("------\n");
      PRINT2M(B,N,N,"%.0f ");  printf("------\n");
      PRINT2M(AB,N,N,"%.0f ");
      */
    }
  else          // worker process
    {
      MPI_Bcast(B,N*N,MPI_DOUBLE,0,MPI_COMM_WORLD); // receive broadcast
      // synchronous receive
      MPI_Recv(A+(rank*interval),interval*N,MPI_DOUBLE,0,rank,
                  MPI_COMM_WORLD,&stat);
      compinterval(rank*interval, interval);
      // send results back to root process, synchronous send
      MPI_Send(AB+(rank*interval),interval*N,MPI_DOUBLE,0,rank,
                        MPI_COMM_WORLD);
    }
  MPI_Finalize();
} //main

void compinterval(int start, int interval)
{
  int i, j, k;
  for (i=start;i<start+interval;i++)
    for(j=0;j<N;j++)
      {
            AB[i][j]=0;
            for(k=0;k<N;k++) AB[i][j] += A[i][k] * B[k][j];
      }
}
```

Пример 2. Перемножение матриц – UPC

```
#include <upc.h>

#define N 4
#define P 4
#define M 4

// a and c are blocked shared matrices
shared [N*P /THREADS] int a[N][P], c[N][M];

shared[M/THREADS] int b[P][M];

int main () {
  int i, j , l; // private variables

  upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
      for (j=0 ; j<M ;j++) {
        c[i][j] = 0;
        for (l = 0; l<P ; l++)
          c[i][j] += a[i][l]*b[l][j];
    }
  }
  return 0;
}
```

Пример 2. Перемножение матриц – UPC (блочная версия)

```
#include <upc_relaxed.h>

#define N 4
#define P 4
#define M 4

// a and c are blocked shared matrices
shared [N*P /THREADS] int a[N][P], c[N][M];
shared[M/THREADS] int b[P][M];

int b_local[P][M]; //local global variable

int main () {
  int i, j , l; // private variables

  upc_memget(b_local, b, P*M*sizeof(int));
  upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
    for (j=0 ; j<M ;j++) {
      c[i][j] = 0;
      for (l= 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b_local[l][j]; // now local
      }
  }
  return 0;
}
```

Советы по оптимизации

- Приватизация пространства
  - Используйте указатели на локальные массивы (private) вместо указателей на shared области памяти
  - Для этого есть средства: casting, присваивание
- Блочные передачи данных
  - Используйте блочные копирования вместо копирования элементов по одному в цикле
- Обязательно учитывайте физической распределение данных по потокам
  - Стремитесь минимизировать коммуникации и доступ к удаленным данным

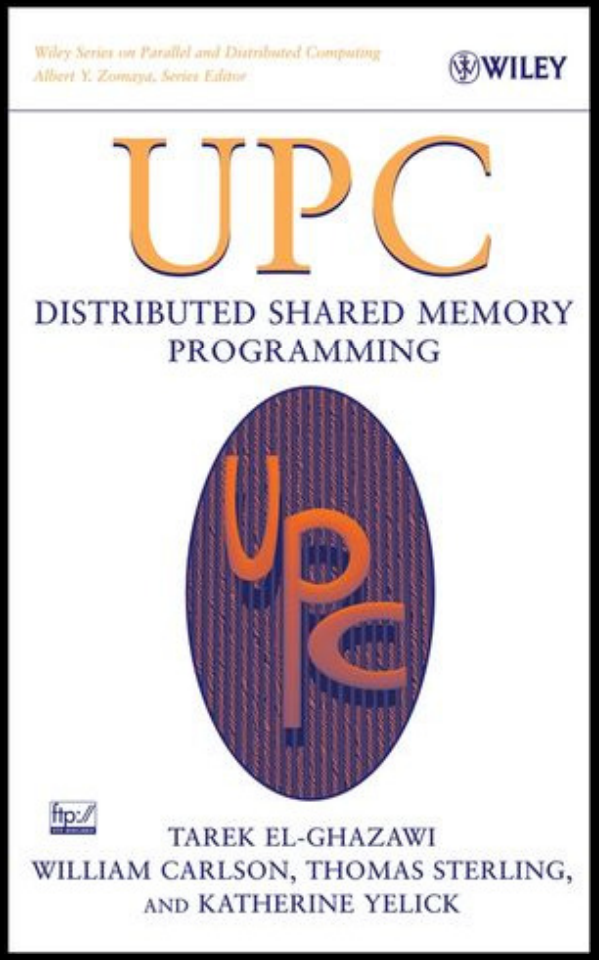
Заключение

- UPC довольно прост в программировании
  - Особенно для C-программистов
  - Особенно по сравнению с другими парадигмами параллельного программирования (MPI, Charm++)
- Производительность UPC сравнима с производительностью MPI
  - На некоторых системах UPC может превосходить MPI
- Вручную оптимизированный код с блочными пересылками данных все же значительно проще, чем MPI
  - Язык и среда выполнения (runtime) берут на себя всю рутинную работу по коммуникации

Вопросы для дальнейшего изучения

- Приведение (casting) указателей
- Синхронизация
  - Неблокирующие барьеры «Split-phase barriers»
  - upc_fence
- Модели консистентности памяти
  - strict
  - relaxed
- Коллективные операции
  - Не описаны стандартом, реализованы в отдельной библиотеке
- Ввод-вывод

Источники информации

1. http://upc.gwu.edu
2. 

UPC: Distributed Shared Memory Programming

Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick

Wiley, 2005

ISBN: 978-0-471-22048-0