High Performance Parallel Computing (3.) Threads OpenMP, UPC

Imre Szeberényi, Éva Geist

BME IIT

<szebi@iit.bme.hu>

MŰEGYETEM 1782

Short summary

- Models of parallel computers
  - Flynn’s taxonomy
  - idealized parallel computer model
- Programming models
  - Shared memory
  - Distributed shared memory
  - Message passing
- Classes (types) of parallel computers
  - computers with vector processors
  - Symmetric Multiprocessors (SMP)
  - Massively Parallel(MPP)
  - Cluster

Tools (languages)

Tools are not connected to hardware architecture, but they efficiency may differ.

- Linda
- Express
- PVM
- MPI

Design of paralell algorithm

- Computing model (ex. PRAM)
  - algorithm efficiency
- Programming model
  - Processing element (process, task, thread, processor, core)
- Systematic design methods (pl. PACM)
- Design Patterns

Design patterns

- Single Program Multiple Data (SPMD)
  - frequent used method
- Master-Worker (master-slave)
  - fault tolerance, load balancing almost automatic
- Pipeline
  - serial coupled processing element
- Divide and Conquer
- Fork – Join

SPMD pattern

- Same as the SIMD on instruction level
- Mostly results of domain decomposition, but also can be used by control-parallel strategy.
- Supported by the most frequently used tools
- The presented examples (pi, prime) based on SPMD model.
- Sometimes implemented by threads.

Threads

- The memory area of the processes are completely separated.
- The memory area of the threads are common except of stack area.
- In this way the context switch is faster and they can communicate each other easily.
  - pthread_create(), pthread_join(), pthread_exit()
  - pthread_mutex_..., pthread_cond_...

pthread example

thread function

```
#include <stdio.h>
#include <pthread.h>
void* do_loop(void *id)
{
    int i, j;
    float f = 0;
    char me = *(char *)id;
    for (i=0; i<10; i++) {
        for (j=0; j<5000000; j++) f++;
        printf("Thread_%c: %d\n", me, i);
    }
    pthread_exit(NULL);
}
```

unique data

pthread example/2

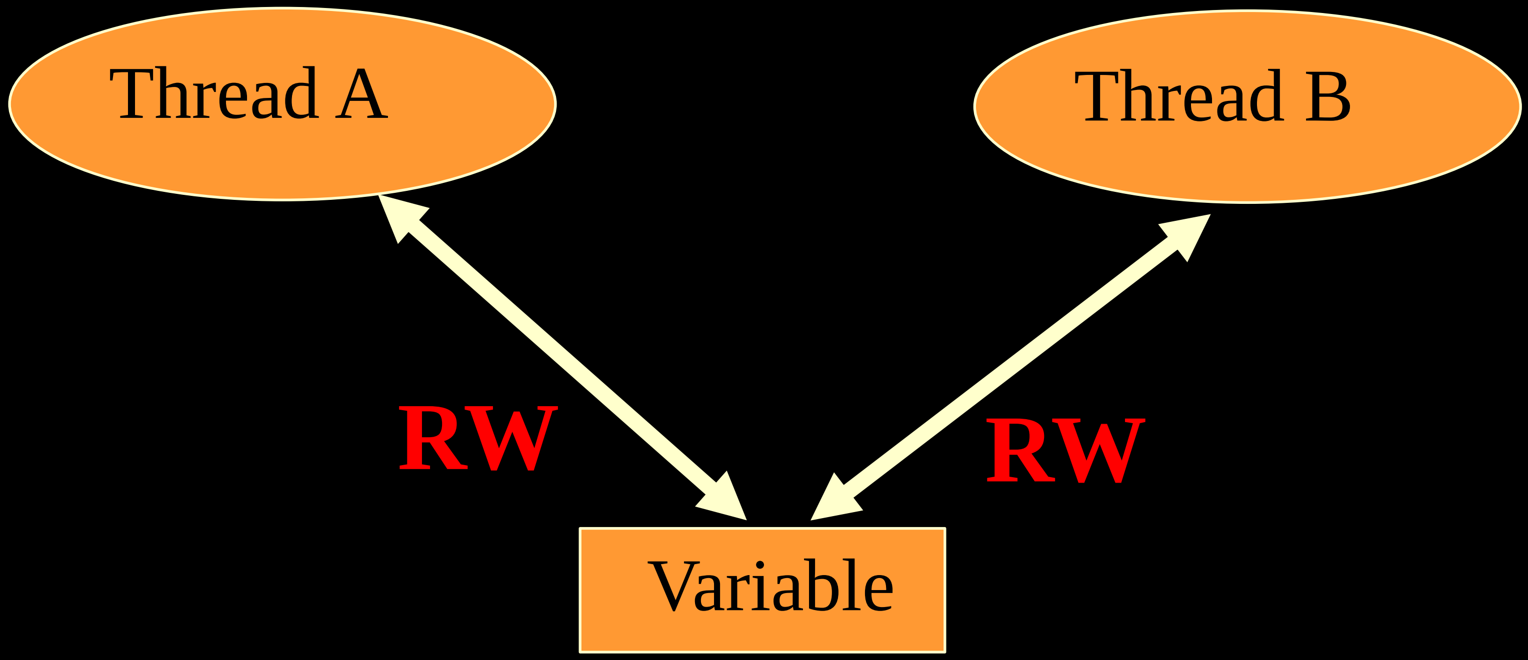
```
int main(int argc, char* argv[])
{
    pthread_t thread_a, thread_
    char a='a', b='b', c='c';

    pthread_create(&thread_a, NULL,
                              do_loop, &a);
    pthread_create(&thread_b, NULL,
                              do_loop, &b)
    do_loop(&c);
    printf(" can't get here!\n")
    return 0;
}
```

thread handle

unique parameter

thread function

Race condition problem



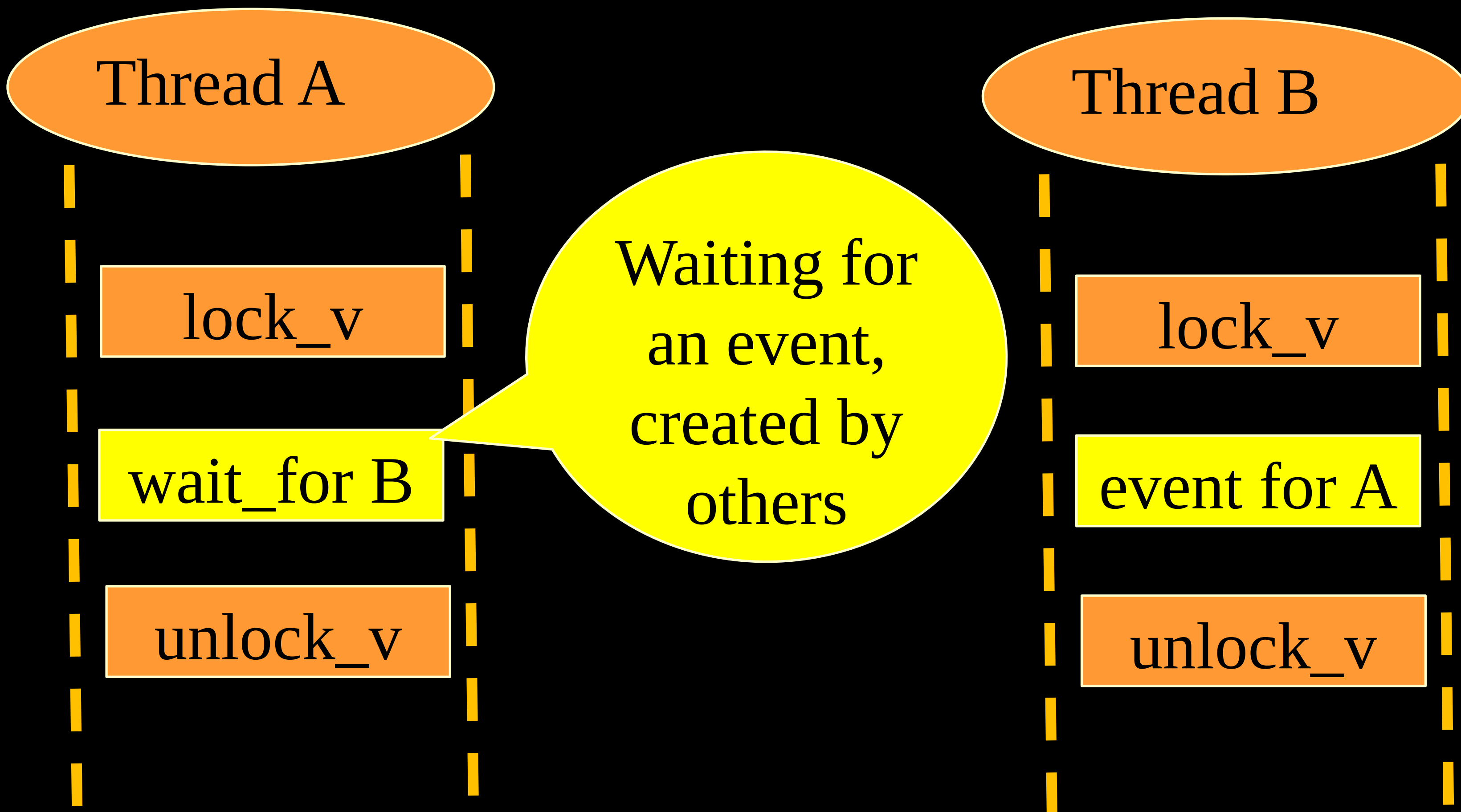
Solution: mutual exclusion

- mutex:
  - pthread_mutex_init
  - pthread_ mutex_destroy
  - pthread_ mutex_lock
  - pthread_ mutex_trylock
  - pthread_ mutex_unlock

pthread_mutex

```
....
pthread_mutex_t mutex_cnt;
....
pthread_mutex_init(&mutex_cnt, NULL);
....
  pthread_mutex_lock(&mutex_cnt);

  // critical region

  pthread_mutex_unlock(&mutex_cnt);
```

Deadlock problem



Solution: conditional variable

- condition:
  - pthtread_ cond_init
  - pthtread_ cond_destroy
  - pthtread_ cond_wait
  - pthtread_ cond_timedwait
  - pthtread_ cond_signal
  - pthtread_ cond_broadcast

pthread_cond

```
pthread_mutex_t mutex_cnt;
pthread_cond_t cond_cnt;
pthread_mutex_init(&mutex_cnt, NULL);
pthread_cond_init(&cond_cnt, NULL);
..// Thread A
  pthread_mutex_lock(&mutex_cnt);
    ....
  // Critical region; waiting for an event
   pthread_cond_wait(&cond_cnt,
  &mutex_cnt);
    ....
  pthread_mutex_unlock(&mutex_cnt);
```

pthread_cond (2)

```
....
.... // Thread B
  pthread_mutex_lock(&mutex_cnt);
      ....
      // The expected event occurs
      pthread_cond_signal(&cond_cnt);
      ....
  pthread_mutex_unlock(&mutex_cnt);
 ....
```

Problems with threads

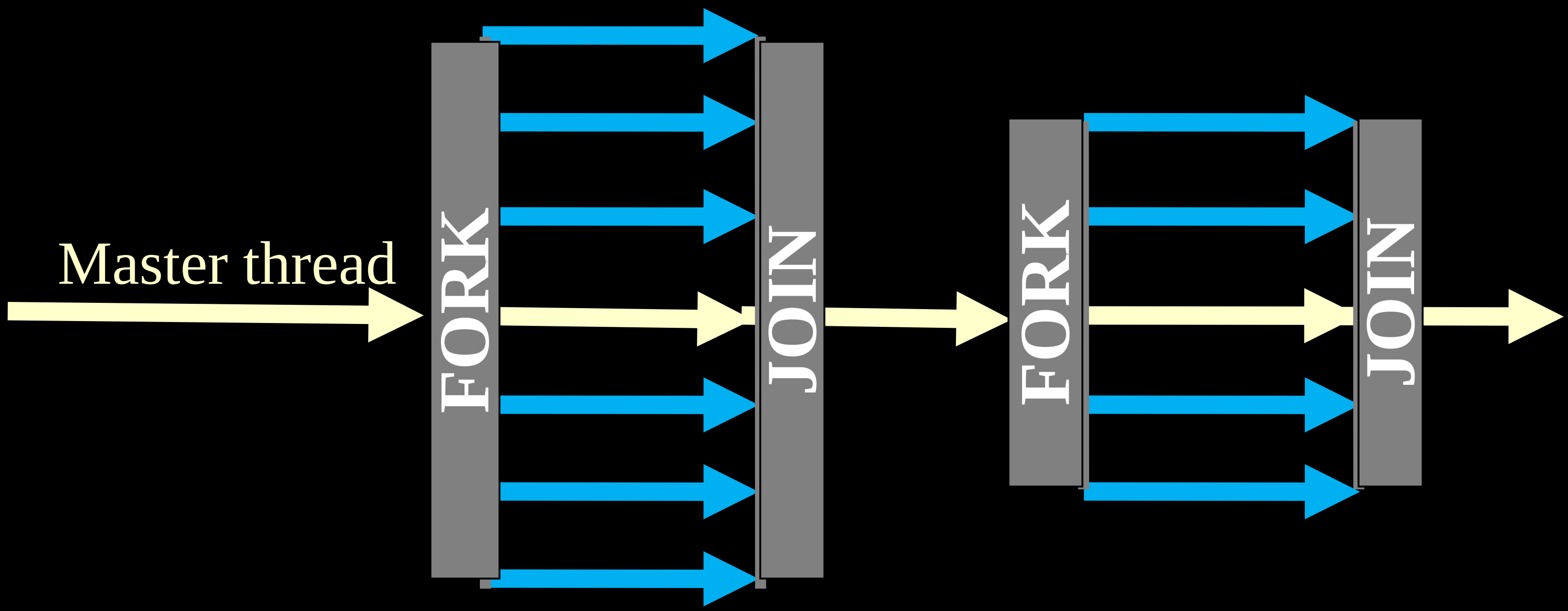
- Implementation dependent
  - UNIX uses least 2 different implementations
  - Windows (MS) other 2
  - Different language bindings
- No portable solution
- Hard to adapt to the real core numbers
- Not an industry standard

OpenMP

- Language annotation
- The programmer can concentrate to the real problem.
- The parallelization only a possibility
- Shared memory model
- Industrial standard

openmp.org,
computing.llnl.gov/tutorials/openMP,
openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf

Execution model



Shared memory modell

- The threads are communicate through variables
- The variable sharing are defined on language level
- Race condition problem solution
  - Synchronisation tools
  - Minimising the shared variable usage

Syntax

- #pragma omp construct [clause [clause] …]
- Refers to a block
- OpenMP constructs:
  - Parallel regions
  - Work sharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

Parallel regions

```
#pragma omp parallel [clause ...]  newline
   structured_block
```

- if (scalar_expression)
- private (list)
- shared (list)
- default (shared | none)
- firstprivate (list)
- reduction (operator: list)
- copyin (list)
- num_threads (integer-expression)

Parallel regions example

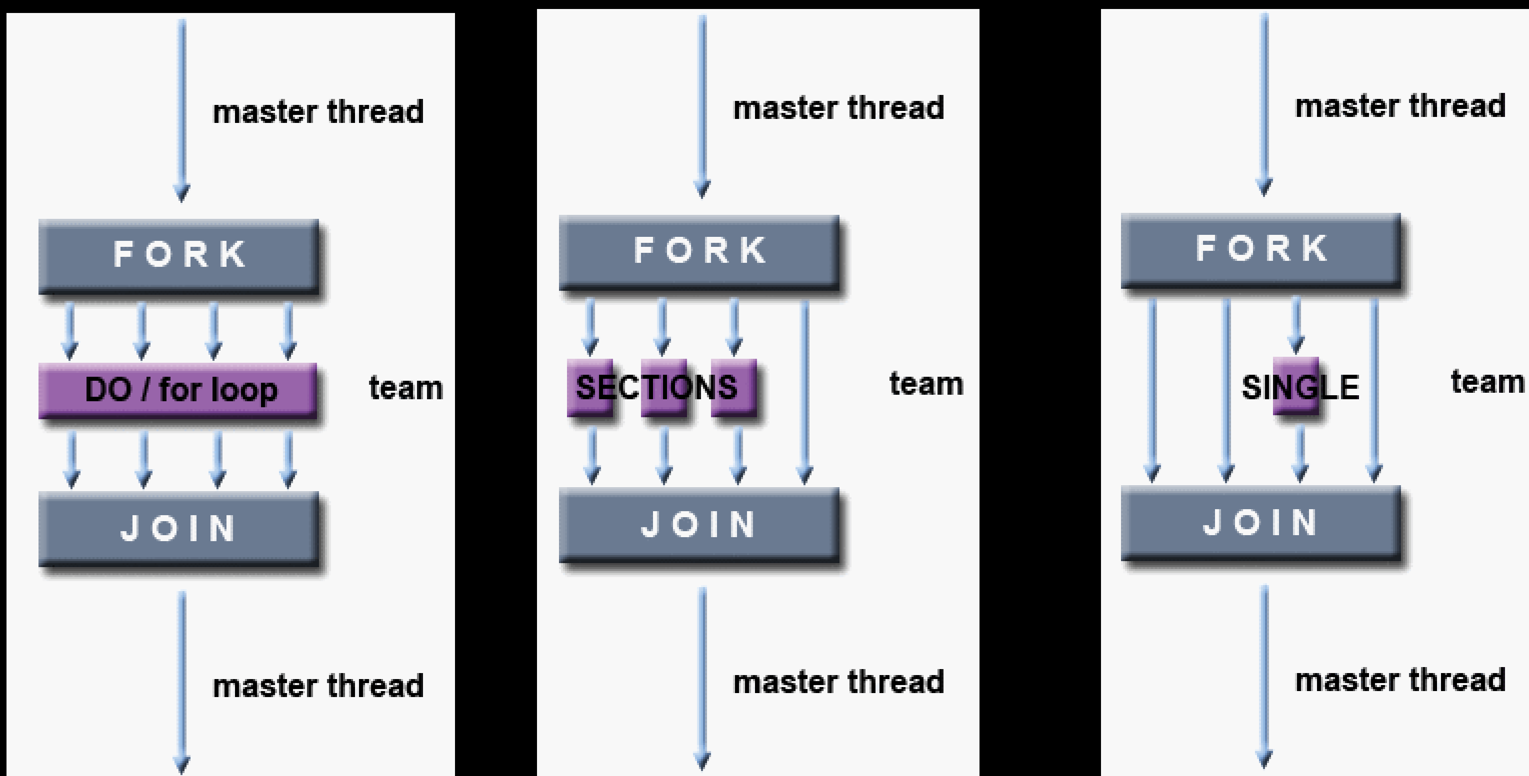
```
double D[1000] = { 1, 2, 3, 4 };
#pragma omp parallel
{
  int i; double sum = 0;
  for (i=0; i<1000; i++) sum += D[i];
  printf(“Thread %d computes %f\n”,
                    omp_get_thread_num(), sum);
}
```

shared

private

// runs as many threads are available

Work sharing



WS: sections

```
#pragma omp sections [clause ...]  newline
{
 #pragma omp section  newline
    structured_block
 #pragma omp section  newline
   structured_block
}
```

private (list)
firstprivate (list)
lastpivate (list)
reduction (operator: list)
nowait

Sections example

```
#pragma omp sections
{
 #pragma omp section
 a = computation_1();
 #pragma omp section
 b = computation_2();
}
c = a + b;
```

Time consuming processes running in parallel

WS: single

```
#pragma omp single [clause ...]  newline
   structured_block
```

```
private (list)
firstprivate (list)
nowait
```

Cycle parallelization by „hand”

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nThr = omp_get_num_threads();
  int istart = id*N/nThr, iend = (id+1)*N/nThr;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

Automatic parallelization

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

Applicable on simple for only:

- only one int loop variable,
- cmp. op: <, >, <=, >
- incrementing/decrementing: ++, --, assigmnet op.
- cycle independent inic, cond, and increment

WS: for

```
#pragma omp for [clause ...]  newline
   for_loop
```

- Schedule (type [,chunk)
- ordered
- private (list)
- firstprivate (list)
- lastprivate (list)
- shared (list)
- reduction (operator: list)
- collapse
- nowait

Scheduling

- schedule(static [, chunksize])
  - static divisions
  - Default: same sizes
  - Round-robin (more slices than threads)
- schedule(dynamic [, chunksize])
  - dynamic slicing
  - Default chunksize = 1
- schedule(guided [, chunksize])
  - Dynamic, exponential

Granulality

- #pragma omp parallel if (expression)
  - condition for parallelization
- #pragma omp num_threads (expression)
  - the number of threads can be modified

Data Environment

- Shared variables

  - global variables

```
int sum = 0;
#pragma omp parallel for
for (int i=0; i<N; i++) sum += i;
```

- Private
  - automatic variables in parallel blokks
  - automatic variables in functions
  - explicit private declaration

Variable types

- private:
  - private instance without initialization
  - same as it would be in {}
- firstprivate:
  - private initialized from the variable
- lastprivate:
  - value of last loop copied back
- threadprivate:
  - persistent private

Variable examples

```
int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { … }
```

```
int idx=1;
int x = 10;
#pragma omp parallel for
#pragma omp firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i]==x) idx = i;
}
```

Variable examples /2

```
int data[100];
#pragma omp threadprivate(data)
…
#pragma omp parallel for copyin(data)
for (int i=0; i<n; i++)
        data[i]++;
```

Reduction

```
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i =0; i<N; j++)
  sum = sum+a[i]*b[i];
```

- Only skalar variable
- each thread has a temporary
- the form is: x op expr
- x++, ++x, x--, --x,
- op could not been overloaded

Syncronization

- Single/Master execution

  #pragma omp single

  #pragma omp master

- Critical sections, Atomic updates

  #pragma omp critical [name]

  #pragma omp atomic

  update_statement

  only scalar

Syncronization /2

- Ordered

  #pragma omp ordered

```
int vec[100];
#pragma omp parallel for ordered
for (int i=0; i<100; i++) {
  vec[i] = 2 * vec[i];
  #pragma omp ordered
  printf(“vec[i] = %d\n”, vec[i]);
}
```

Syncronization /3

- Barriers

  #pragma omp barrier

- Nowait

  #pragma omp sections nowait

- Flush

  #pragma omp flush (list)

- Reduction

Summary of directives

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Controll functions

- omp_set_dynamic(int)/ omp_get_dynamic()
- omp_set_num_threads(int)/ omp_get_num_threads()
  - OMP_NUM_THREADS env.
- omp_get_num_procs()
- omp_get_thread_num()
- omp_set_nested(int)/omp_get_nested()
- omp_in_parallel()
- omp_get_wtime()
- omp_init_lock(), omp_destroy_lock(),
- omp_set_lock(), omp_unset_lock(),
- omp_test_lock()

Unified Parallel C (UPC)

- extesion of C99
- Shared memory model
- supports the NUMA
- development started in 1999.
- most supercomputers support it.

upc-lang.org, upc.lbl.gov,
upc.gwu.edu/tutorials/UPC-SC05.pdf

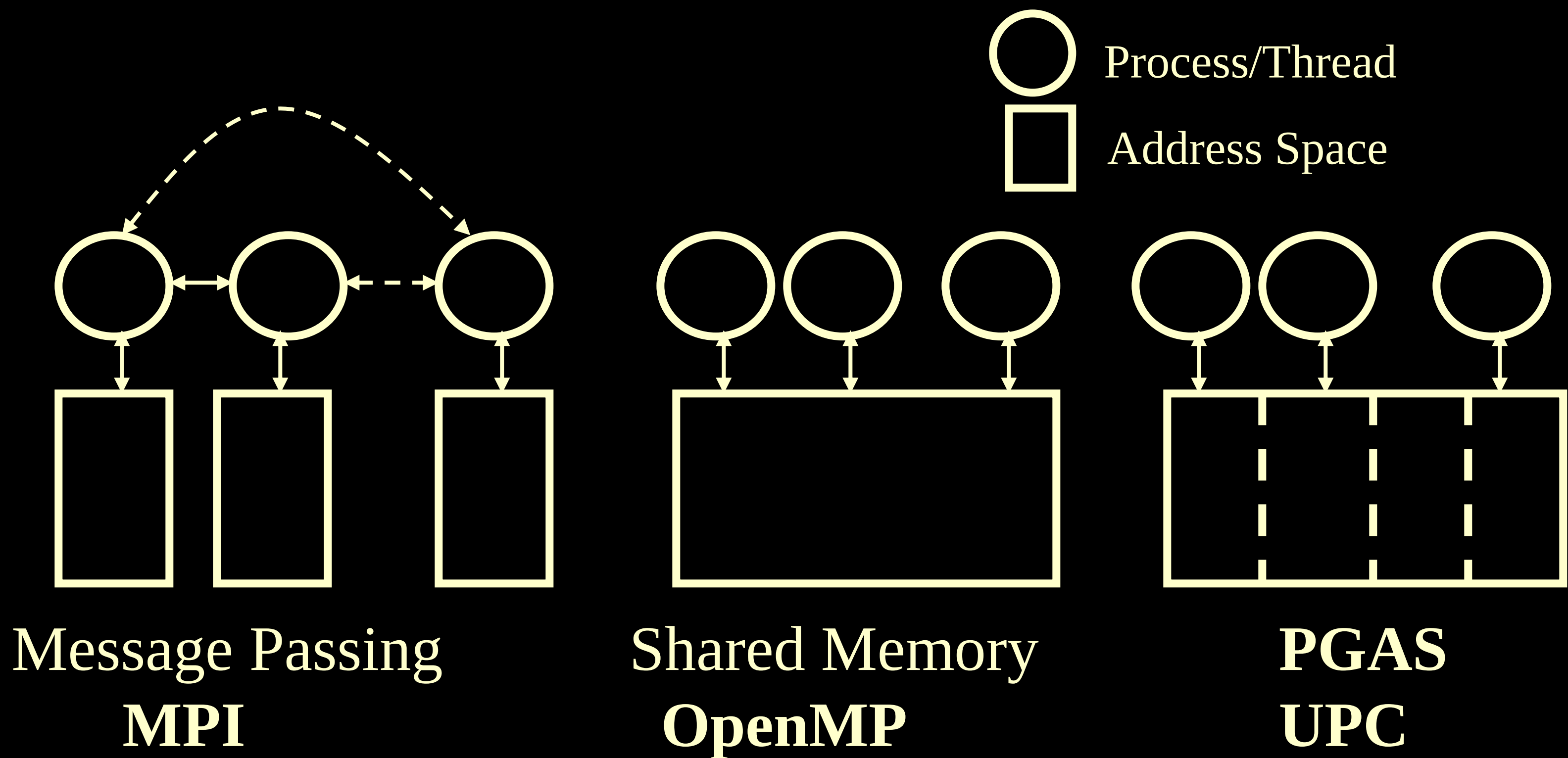
Main features of UPC

- Explicit parallel model
  - Independent threads (thread != OS thread)
  - Each thread complies to the C standard (many main).
- Partitioned Global Address Space(PGAS)
  - shared and private data
  - scalars are in the p0, but arrays are distributed
- Synchronization primitives
- Runtime library
  - scatter, gather, reductions, exchange, ..

UPC keywords

- MYTHREAD, THREADS
- shared, realaxed, strict,
- upc_barrier, upc_notify, upc_wait
- upc_forall,
- upc_blokcsizeof, upc_elementsizeof, upc_localsizeof, UPC_MAXB_LOCK_SIZE
- upc_fence,

Different memory models



Partitioned Global Address Sp.

- The shared arrays are distributed in the memory associated to the processing elements (affinity)
- The threads can work more efficiently on the „nearest” data.

```
shared int a[100];  // cyclic distribution
upc_forall(int i = 0; i < 100; i++; &a[i]) …
// The given cycle run on the nearest data
```

Affinity and block size

```
shared [block_size] type array[N];
affinity = (i/block_size)%THREADS
shared [3] int A[4][THREADS];
supposed: THREADS = 4
```

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

example: PI

```
shared long hits[THREADS];    // counters
....
upc_forall (i=0; i < my_trials; i++; continue)
   hits[MYTHREAD] += isInside();
upc_barrier;                  // wait for all

if (MYTHREAD == 0) // summ of the local cnts
   for (i = 1; i < THREADS; i++) hits[0] += hits[i];
```

Examples

para.iit.bem.hu/~szebi/para/threads

Para.iit.bem.hu/~szebi/para/OpenMP

Para.iit.bem.hu/~szebi/para/UPC

 # For using UPC compiler the environment
 # should be set first:

Module load upc