Concepts of Parallel Programming Languages

Christoph Kessler, IDA

Parallel Language Concepts

- **Parallel control flow**
  - Fork-join style parallelism, SPMD style parallelism
  - Nested parallelism
  - Parallel loops, Sections
  - Parallel loop scheduling
  - Implicit parallelism
- **Synchronization & Consistency**
  - Futures
  - Supersteps and Barriers
  - Array assignments
  - Fence / Flush
  - Semaphores & Monitors
  - Atomic operations
  - Transactions
- **Address space**
  - Global Address Space, Sharing
  - Pointer models
  - Tuple space
- **Data locality & mapping control**
  - Co-Arrays
  - Virtual topologies
  - Alignment, distribution, mapping
  - Data distributions
  - Data redistribution
- **Communication**
  - Collective communication
  - One-sided communication

    (see earlier lecture on MPI)

Some parallel programming languages

(partly) covered here:

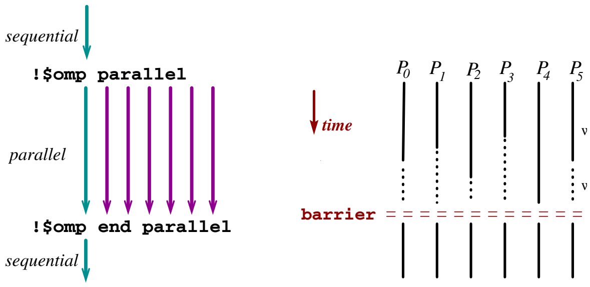
- Fork (see earlier lecture)
- Cilk (see earlier lecture)
- MPI (see earlier lecture)
- OpenMP
- HPF
- UPC / Titanium
- NestStep
- ZPL
- ...

Relationship between parallel and sequential programming languages

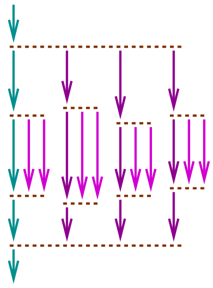
- Big issue: Legacy code in Fortran, C, (C++)
  - Practically successful parallel languages must be interoperable with, and, even better, syntactically similar to one of these
- Compliance with sequential version is useful
  - e.g. C elision of a Cilk program is a valid C program doing the same computation
  - OpenMP
- Incremental parallelization supported by directive-based languages
  - e.g. OpenMP, HPF

Parallel Control Flow

Fork-Join-Style Parallelism vs. SPMD-Style Parallelism



Nested Parallelism

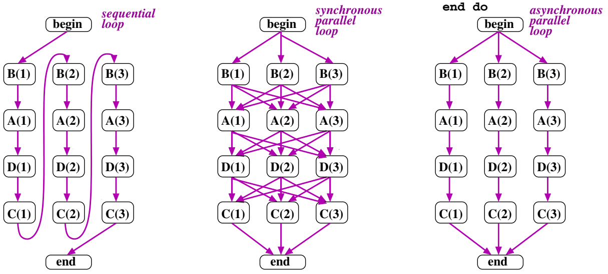


Parallel Loop Constructs

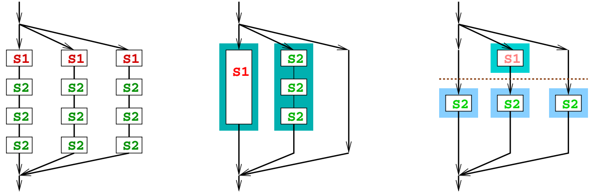
```
do i = 1, 3
  A(i) = B(i)
  C(i) = D(i)
end do
```

```
forall  i = 1, 3
  A(i) = B(i)
  C(i) = D(i)
end forall
```

```
!hpf$ independent
do i = 1, 3
  A(i) = B(i)
  C(i) = D(i)
end do
```



Parallel Sections



```
!$OMP PARALLEL


CALL S1(...)


DO I=1,3
    CALL S2(...)
END DO

!$OMP END PARALLEL
```

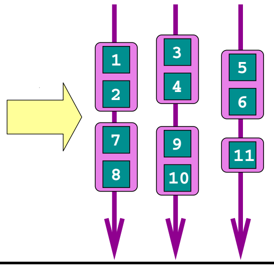
```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
CALL S1(...)

!$OMP SECTION
DO I=1,3
    CALL S2(...)
END DO
!$OMP SECTIONS
!$OMP END PARALLEL
```

```
!$OMP PARALLEL

!$OMP SINGLE
CALL S1(...)
!$OMP END SINGLE
!$OMP DO
DO I=1,3
    CALL S2(...)
END DO
!$OMP END DO
!$OMP END PARALLEL
```

Parallel Loop Scheduling (1)

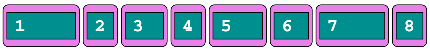
- Static scheduling
  - Chunk scheduling

```
!$omp do schedule ( STATIC, 2 )
      do i = 1, ..., 11
         ....
      end do
```

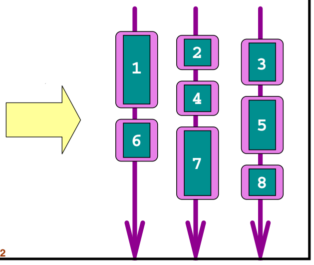


Parallel Loop Scheduling (2)

- Dynamic Loop Scheduling
  - Chunk Scheduling

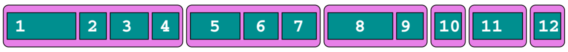


```
!$omp do schedule ( DYNAMIC, 1 )
      do i = 1, ..., 8
         ....
      end do
```



Parallel Loop Scheduling (3)

- Guided Self-Scheduling
  - Chunk scheduling



```
!$omp do schedule ( GUIDED, 1 )
      do i = 1, ..., 12
         ....
      end do
```

Parallel Loop Scheduling (4)

- **Affinity-based Scheduling**
  - Dynamic scheduling, but use locality of access together with load balancing as scheduling criterion
  - ”cache affinity”
- Example: UPC forall loop
  -
    ```
    shared float x[100], y[100], z[100];
    ...
    upc_forall ( i=0;  i<100;  i++;  &x[i] )
       x[i] = y[i] + z[i];
    ```

    expression describing affinity
  - Iteration i with assignment x[i] = y[i] + z[i] will be performed by the thread storing x[i], typically (i % THREADS)

FDA125 Advanced Parallel Programming

Synchronization and Consistency

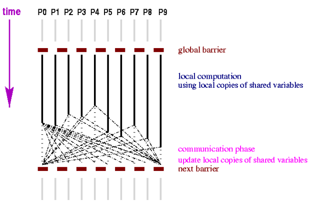
Christoph Kessler, IDA,
Linköpings universitet, 2007.

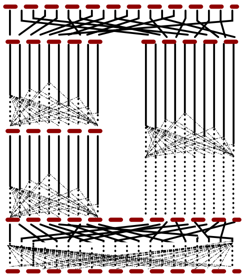
Futures

- A ***future call*** by a thread T1 starts a new thread T2 to calculate one or more values and allocates a ***future cell*** for each of them.
- T1 is passed a read-reference to each future cell and continues immediately.
- T2 is passed a write-reference to each future cell
- Such references can be passed on to other threads
- As (T2) computes results, it writes them to their future cells.
- When any thread touches a future cell via a read-reference, the read stalls until the value has been written.
- A future cell is written only once but can be read many times.
- Used e.g. in Tera-C [Callahan/Smith’90], ML+futures [Blelloch/Reid-Miller’97], StackThreads/MP [Taura et al.’99]

Supersteps

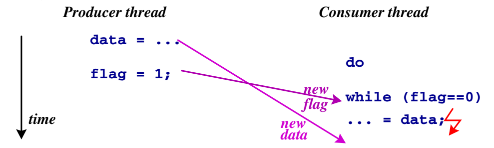
- BSP model: Program executed in series of supersteps
- Nestable supersteps
  - PUB library [Bonorden et al.’99], NestStep [K.’99]





No mutual exclusion possible!

Fence / Flush

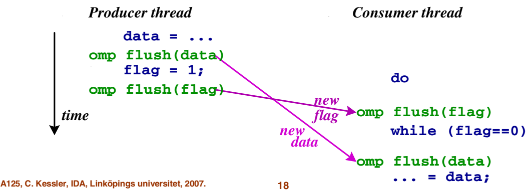


Producer thread:

```
data = ...

flag = 1;
```

Consumer thread:

```
do

while (flag==0)
... = data;
```



Producer thread:

```
    data = ...
omp flush(data)
    flag = 1;
omp flush(flag)
```

Consumer thread:

```
    do

omp flush(flag)
    while (flag==0)

omp flush(data)
    ... = data;
```

Atomic Operations

- Atomic operations on a single memory word
  - SBPRAM / Fork mpadd etc.
  - OpenMP `atomic` directive for simple updates (x++, x--)
  - test&set, fetch&add, cmp&swap, atomicswap ...

Atomic Transactions

- For atomic computations on multiple shared memory words
- Abstracts from locking and mutual exclusion
  - coarse-grained locking does not scale
  - declarative rather than hardcoded atomicity
  - enables lock-free concurrent data structures
- Transaction either commits or fails
- Variant 1: atomic { ... } marks transaction
- Variant 2: special transactional instructions
  e.g. LT, LTX, ST; COMMIT; ABORT
- Speculate on atomicity of non-atomic execution
  - Software transactional memory
  - Hardware TM, implemented e.g. as extension of cache coherence protocols [Herlihy,Moss'93]

Atomic Transactions Example: Lock-based vs. Transactional Map based Data Structure

```
class LockBasedMap
   implements Map
{
   Object mutex;
   Map m;

   LockBasedMap (Map m) {
      this.m = m;
      mutex = new Object();
   }

   public Object get() {
      synchronized (mutex) {
         return m.get();
      }
   }

   // other Map methods. . .
}
```

```
class AtomicMap
    implements Map
{
    Map m;

    AtomicMap (Map m) {
       this.m = m;
    }

    public Object get() {
       atomic {
          return m.get();
       }
    }

    // other Map methods. . .
}
```

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.

Example: Thread-safe composite operation

- Move a value from one concurrent hash map to another
- Threads see each key occur in exactly one hash map at a time

```
void move (Object key) {
   synchronized (mutex ) {
      map2.put ( key, map1.remove(key));
   }
}
```

Requires (coarse-grain) locking
(does not scale)

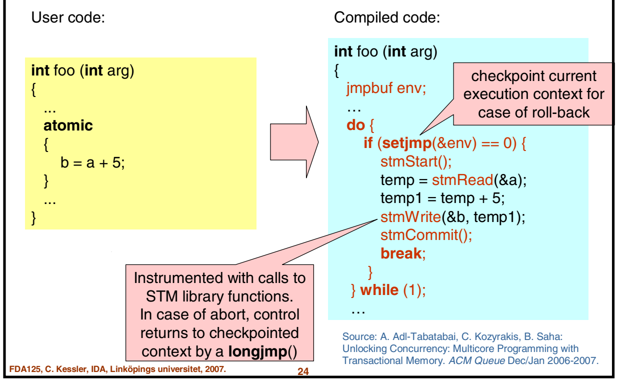
or rewrite hashmap for fine-grained locking
(error-prone)

```
void move (Object key) {
   atomic (mutex ) {
      map2.put (key, map1.remove( key));
   }
}
```

Any 2 threads can work in parallel
as long as different hash table buckets are accessed.

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.

Software Transactional Memory



User code:

```
int foo (int arg)
{
   ...
   atomic
   {
      b = a + 5;
   }
   ...
}
```

Compiled code:

```
int foo (int arg)
{
  jmpbuf env;
  …
  do {
     if (setjmp(&env) == 0) {
        stmStart();
        temp = stmRead(&a);
        temp1 = temp + 5;
        stmWrite(&b, temp1);
        stmCommit();
        break;
     }
  } while (1);
  …
```

checkpoint current execution context for case of roll-back

Instrumented with calls to STM library functions. In case of abort, control returns to checkpointed context by a **longjmp**()

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.

Transactional Memory

- Good introduction:

  A. Adl-Tabatabai, C. Kozyrakis, B. Saha:
  Unlocking Concurrency: Multicore Programming with
  Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.

- More references:

  See course homepage – list of papers for presentation

FDA125 Advanced Parallel Programming

Address space

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Tuple space

- Linda [Carriero, Gelernter 1988]
- Tuple space
  - Associative memory storing data records
  - Physically distributed, logically shared
  - Atomic access to single entries: **put**, **get**, **read**, ...
  - Query entries by pattern matching
    get (”task”, &task_id, args, &argc, &producer_id, 2);
- Can be used to coordinate processes
  - E.g., task pool for dynamic scheduling
  - E.g., producer-consumer interaction

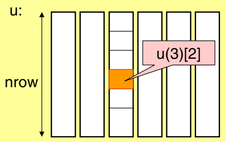
Data Locality Control

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Co-Arrays

- Co-Array Fortran [Numrich / Raid ’98]
- **Co-Arrays**
  - Distributed shared arrays with a **co-array dimension** spanning the processors in a SPMD environment
  - arr(j)[k] – addresses processor k’s copy of arr(j)
  - x(:) = y(:)[q]

Co-Array Fortran Example



```
subroutine laplace ( nrow, ncol, u )
  integer, intent(in)   ::  nrow, ncol
  real, intent(inout)   ::  u(nrow) [*]
  real                  ::  new_u(nrow)
  integer               ::  i, me, left, right

  new_u(1) = u(nrow) + u(2)
  new_u(nrow) = u(1) + u(nrow-1)
  new_u(2:nrow-1) = u(1:nrow-2) + u(3:nrow)
  me = this_image(u)   ! Returns the co-subscript within u
                       ! that refers to the current image
  left = me-1;
  if (me == i) left = ncol
  right = me + i;
  if (me == ncol) right = 1
  call sync_all( (/left,right/) )  ! Wait if left and right have not already reached here
  new_u(1:nrow) = new_u(1:nrow) + u(1:nrow) [left] + u(1:nrow) [right]
  call sync_all( (/left,right/) )
  u(1:nrow) = new_u(1:nrow) - 4.0 * u(1:nrow)
end subroutine laplace
```

Source: Numrich, Reid: *Co-Array Fortran for parallel programming.* Technical report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, 1998.

Virtual topologies

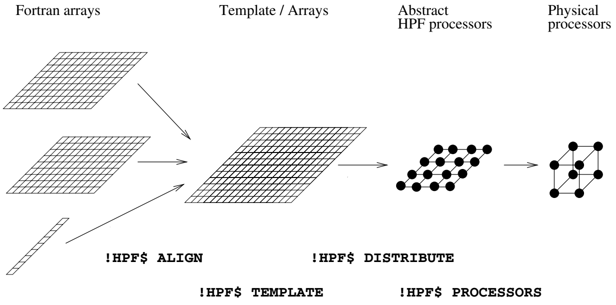
Example: arrange 12 processors in 3x4 grid:

```
int dims[2], coo[2], period[2], src, dest;
period[0]=period[1]=0;  // 0=grid, !0=torus
reorder=0; // 0=use ranks in communicator,
        // !0=MPI uses hardware topology
dims[0] = 3; // extents of a virtual
dims[1] = 4; // 3X4 processor grid

// create virtual 2D grid topology:
MPI_Cart_create( comm, 2, dims, period,
                 reorder, &comm2 );

// get my coordinates in 2D grid:
MPI_Cart_coords( comm2, myrank, 2, coo );

// get rank of my grid neighbor in dim. 0
MPI_Cart_shift( comm2, 0, +1, // to south,
                &src, &dest); // from south
...
```

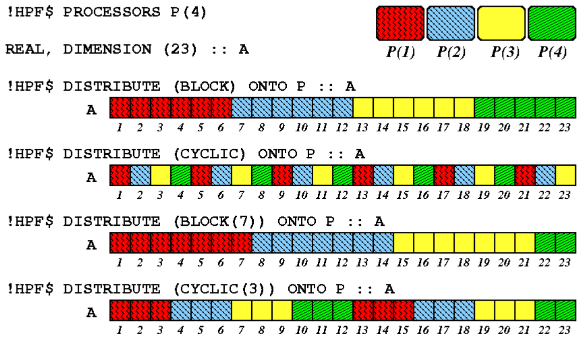
(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

```
...
coo[0]=i; coo[1]=j;

// convert cartesian coordinates
// (i,j) to rank r:
MPI_Cart_rank(comm, coo, &r);

// and vice versa:
MPI_Cart_coords(comm,r,2,coo);
```

HPF Mapping Control:
Alignment, Distribution, Virtual Processor Topology

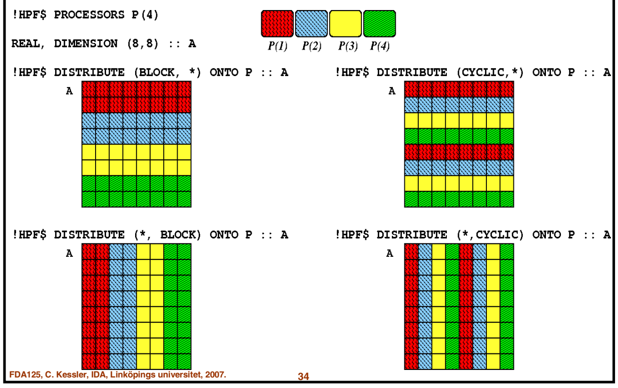


Data Distribution (1)



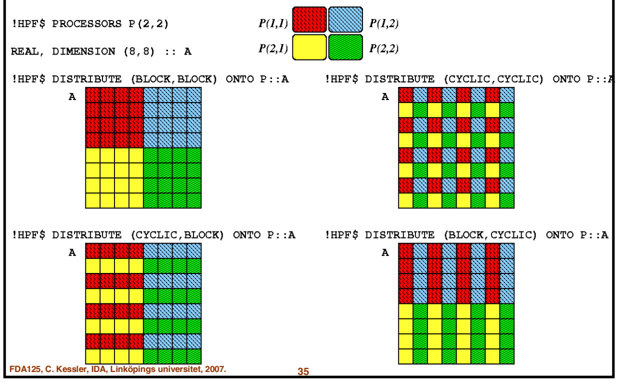
```
!HPF$ PROCESSORS P(4)

REAL, DIMENSION (23) :: A

!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A

!HPF$ DISTRIBUTE (CYCLIC) ONTO P :: A

!HPF$ DISTRIBUTE (BLOCK(7)) ONTO P :: A

!HPF$ DISTRIBUTE (CYCLIC(3)) ONTO P :: A
```

Data Distribution (2)



```
!HPF$ PROCESSORS P(4)

REAL, DIMENSION (8,8) :: A

!HPF$ DISTRIBUTE (BLOCK, *) ONTO P :: A

!HPF$ DISTRIBUTE (CYCLIC,*) ONTO P :: A

!HPF$ DISTRIBUTE (*, BLOCK) ONTO P :: A

!HPF$ DISTRIBUTE (*,CYCLIC) ONTO P :: A
```

Data Distribution (3)



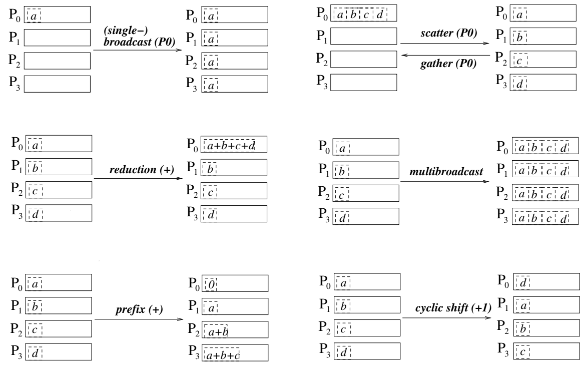
```
!HPF$ PROCESSORS P(2,2)

REAL, DIMENSION (8,8) :: A

!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P::A

!HPF$ DISTRIBUTE (CYCLIC,CYCLIC) ONTO P::A

!HPF$ DISTRIBUTE (CYCLIC,BLOCK) ONTO P::A

!HPF$ DISTRIBUTE (BLOCK,CYCLIC) ONTO P::A
```

FDA125 Advanced Parallel Programming

Communication

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Collective Communication



Encapsulation of communication context

- Example: MPI Communicator
- Needed for parallel components