Parallel Programming Models

Lecture 3
Introduction to High Performance Computing
IN4049 TUDelft
2013

Recap Lecture 2 [1]

- Design parallel applications
  - *Systematic process*
- Performance evaluation using Amdhal's law
  - *First estimates are available at design time!*
- Models of parallel architectures
  - *Shared memory*
  - *Distributed memory*
  - *Virtual shared memory*

Recap Lecture 2 [2]

Models of parallel computation

- Conceptual level: closer to the application
  - *farmer/worker*
  - *divide and conquer*
  - *data parallelism*
  - *task parallelism*
  - *bulk synchronous*
- System level: closer to the machine
  - *(Logical) data spaces and programs*
  - *Communication and synchronization*

Today – Lecture 3

- Parallel programming models
  - Classic: OpenMP, HPF, MPI
  - Low-level: pthreads
  - Modern reincarnations: TBB, ArBB, Cilk (Satin), Titanium, UPC, MapReduce (lecture 4)
  - HW-centric: CUDA, OpenCL, OpenACC, … (lecture 4)
- Reading assignment (optional): on comparing performance models:
  - *Varbanescu et. al - "Towards an Effective Unified Programming Model for Many-Cores", IPDPS Workshops 2012*

Parallel programming models

Parallel Programming Model

- Programming model = language + libraries that create a model of computation (i.e., abstract machine)
- Abstraction level = qualifier for the distance between the abstract and the real machine
  - *E.g. Assembly language: very low vs. Java: high/very high*
  - *More difficult to estimate for parallel models*
- Empirical evaluation:
  - *High-level languages:*
    - *Pro: Programmer friendly*
    - *Con: Performance impacted by overheads*
  - *Low-level languages:*
    - *Pro: Performance*
    - *Con: Difficult to learn and use*

Parallel Programming Models

- Control model
  - Parallel operations creation
  - (Logical) Ordering
- Data model
  - Shared vs. Private data
  - Logical access to data (i.e., communication)
- Synchronization model
  - Coordinating parallelism
- Cost model(/performance impact)
  - Expensive vs. Cheap operations
  - Overhead

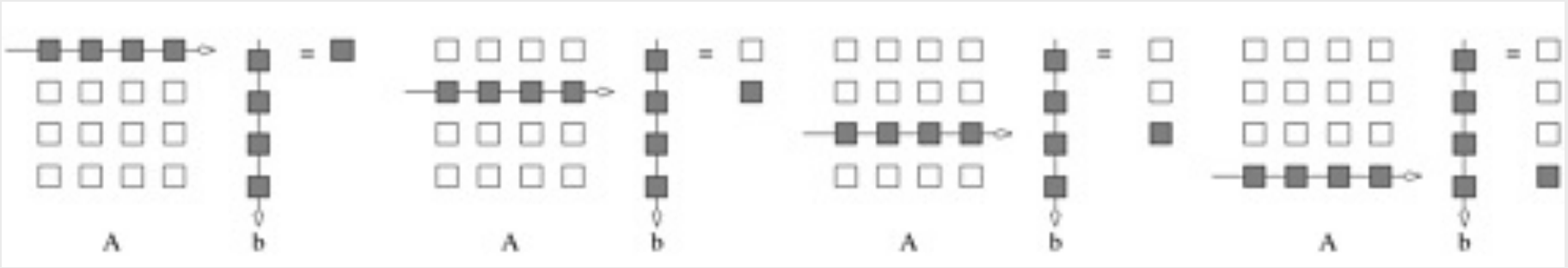
Classical models: OpenMP

App: Matrix-Vector Product (MVP)

- On the forthcoming slides this operation is used for demonstration of the different parallel languages

```
for(i=0; i<dim; i++) {
    y[i]=0;
    for(j=0; j<dim; j++) {

    y[i]=y[i] + A[i,j] * x[j];
    }
}
```



Shared memory: OpenMP

- Control:
  - *Parallel Loops*
  - *Parallel Sections*
- Data:
  - *Shared and/or private, explicitly specified*
- Synchronization
  - *Explicit, using critical sections*
  - *Atomic operations*
- Communication
  - *Shared variables*

OpenMP format

- Extension to C, C++, Fortran (pragma-based)
- General format: `#pragma omp directive [clauses]`
- Number of threads: compile-time / run-time
- Example*

```
1.   int main (int argc, char *argv[]) {
2.   int nthreads, tid;
3.   #pragma omp parallel private(nthreads, tid){
4.     tid = omp_get_thread_num();
5.     printf("Hello World from %d\n", tid);
6.     nthreads = omp_get_num_threads();
7.     if (tid == 0)
8.        printf(“Threads = %d\n", nthreads);
9.   }
```

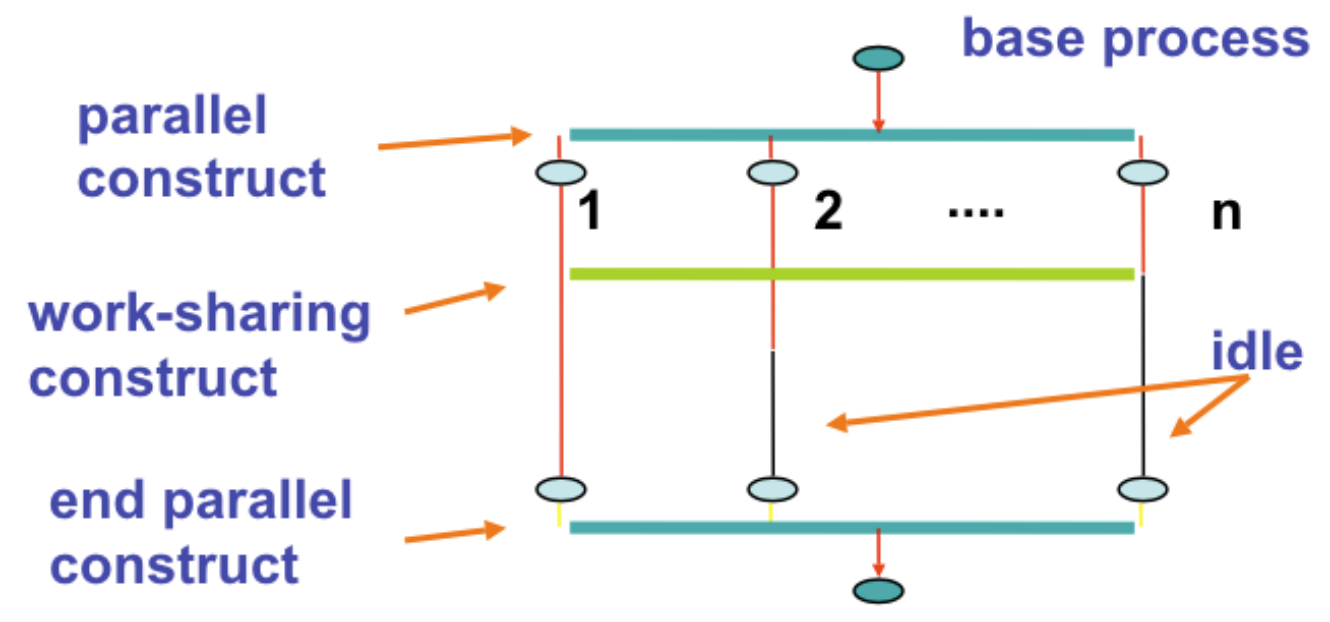
*More: https://computing.llnl.gov/tutorials/openMP/exercise.html

OpenMP execution model

`Parallel` construct creates a number of worker processes (or threads)

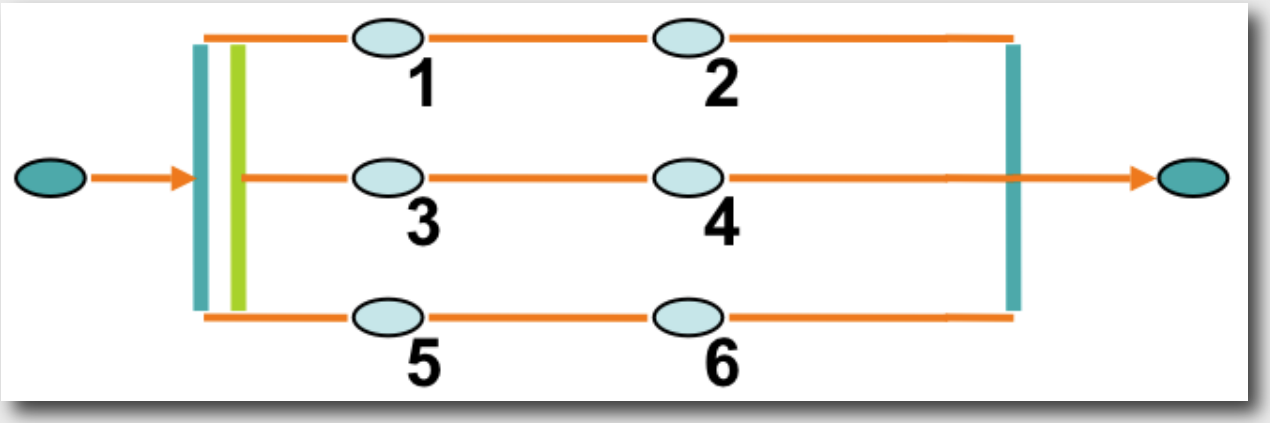
Work-sharing construct hands out work to available workers (threads)

- #pragma omp sections [clauses]
- #pragma omp for [clauses]



OpenMP: parallel for

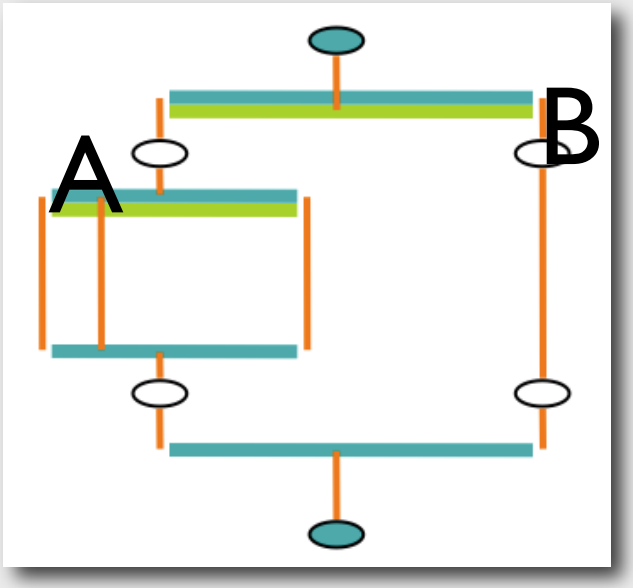

```
#pragma omp parallel
  #pragma omp for schedule(static,2)
     for(i=0;i<dim;i++) { ... }
```



OpenMP: parallel section

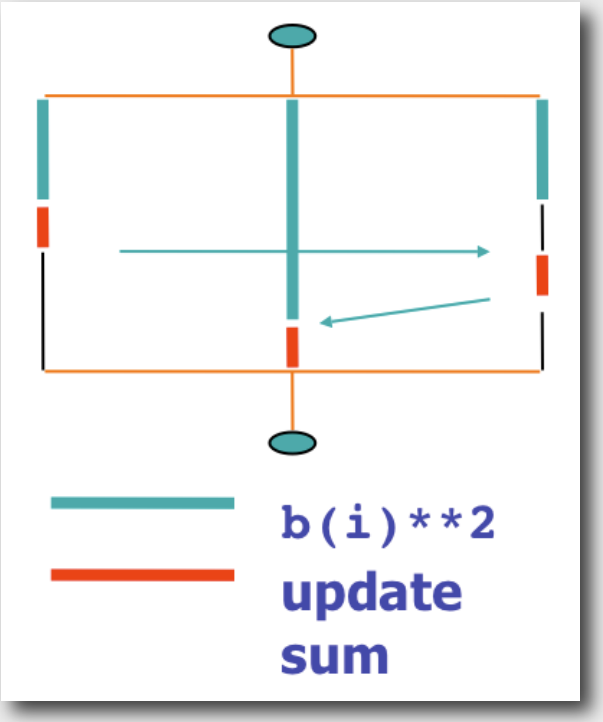
- Combines parallel and work-sharing construct
- Allows independent execution of subprograms
  - *Task parallelism*

```
#pragma omp parallel sections {
   #pragma omp section {
      task_A();       // parallel
for
  }
  #pragma omp section  {
      task_B();       // simple
  }
}
```



OpenMP: locks and critical sections


```
for(i=0;i<dim;i++) {
      sum = sum + b[i]**2;
}
```




```
#pragma omp parallel private(i) reduction(+:sum)
   schedule(dynamic,1) {
     for(i=0;i<dim;i++){ sum = sum + b[i]**2     }
}
```

OpenMP: MVP example


```
for(i=0; i<dim; i++) {
    y[i]=0;
    for(j=0; j<dim; j++) {
     y[i]=y[i] + A[i,j] * x[j];
    }
}
```

```
#pragma omp parallel for private(j)
for(i=0; i<dim; i++) {
    y[i]=0;
    for(j=0; j<dim; j++) {
            y[i]=y[i] + A[i,j] * x[j];
    }
}
```

OpenMP: MVP example


```
for(i=0; i<dim; i++) {
    y[i]=0;
    for(j=0; j<dim; j++) {
     y[i]=y[i] + A[i,j] * x[j];
    }
}
```

```
for(i=0; i<dim; i++) {
    tmp=0;
    #pragma omp parallel for reduction(+ : tmp)
    for(j=0; j<dim; j++) {
      tmp=tmp + A[i,j] * x[j];
    }
    y[i] = tmp;
}
```

OpenMP: Summary

- Shared memory programming model
- Explicit parallelism
  - Fork-join model
- Implicit data distribution
- Allows for data and task parallelism
- Shared and private variables
  - Communication via shared variables
- Synchronization by critical sections
- High abstraction level
- Preserves sequential code (when possible/needed)

OpenMP Quiz

Which of the following *conceptual programming models* be represented in OpenMP:

- *farmer/workers*
- *divide and conquer*
- *data parallelism*
- *task parallelism*
- *bulk synchronous*

- Write a short (pseudo-code) example for each one of those for which this is possible.
- Indicate the functionality.
- Syntax correctness is not an issue.

Low-level model: pthreads

Shared memory: pthreads

- Applications are divided into threads
- Each thread receives its own functionality
  - SPMD – Single process, multiple data

```
for (i=0; i<N; i++)  {
   create_thread(i, function[i]);
   // run_thread(i)
   join_thread(i);  }
```

  - MPMD – Multiple process, multiple data

```
create_thread(1, function_1);
create_thread(2, function_2);

join_thread(1); join_thread(2);
```

pThreads: brief summary

- Thread management is explicit
  - Creation, joining, killing
- Thread scheduling is (typically) done by OS
  - No pre-determined model
- Communication
  - Explicit via shared variables
  - Protection for race conditions needed
  - Protection against deadlocks
- Synchronization
  - Locks
  - Barriers (not available in the standard!)

Pthreads Quiz

Are the any models that cannot be expressed in OpenMP, but can be expressed in pThreads?

- *farmer/workers*
- *divide and conquer*
- *data parallelism*
- *task parallelism*
- *bulk synchronous*

Classical models: HPF

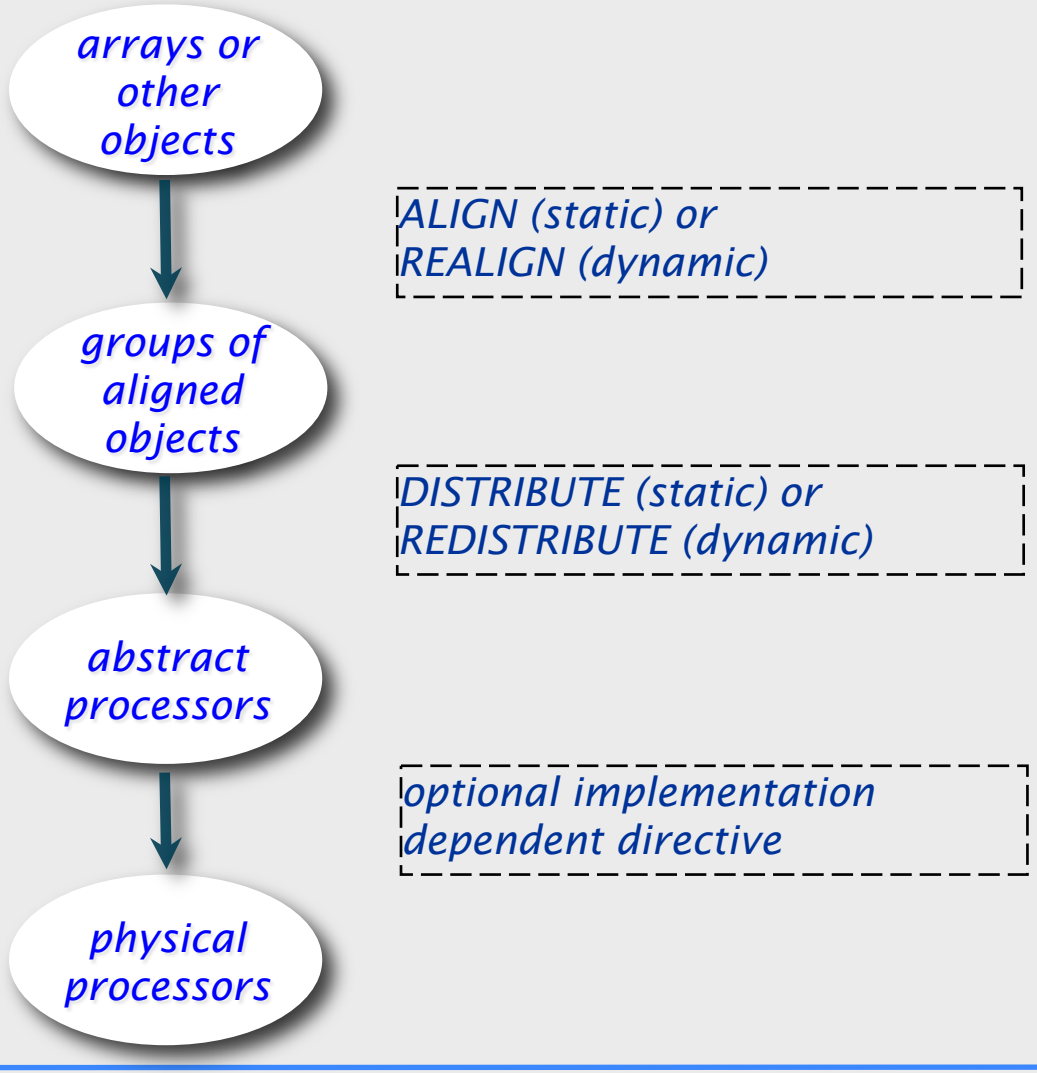
(Logically) Distributed memory: HPF

- High Performance Fortran
- Based on adding annotations to sequential Fortran90 base program
- Main features
  - *parallel constructs*
  - *work-sharing based on data distribution statements*
- Data parallel model
  - *Distributes the array elements over virtual processors*

HPF: features

- Control:
  - *Data-driven: computation follows data distribution*
  - *Parallel arrays and loops*
- Data:
  - *Distributed memory model*
  - *Full arrays still available for the applications, thus ...*
- Synchronization and communication
  - *Implicit*
  - *Generated whenever non-local data is accessed.*

HPF: data distribution model

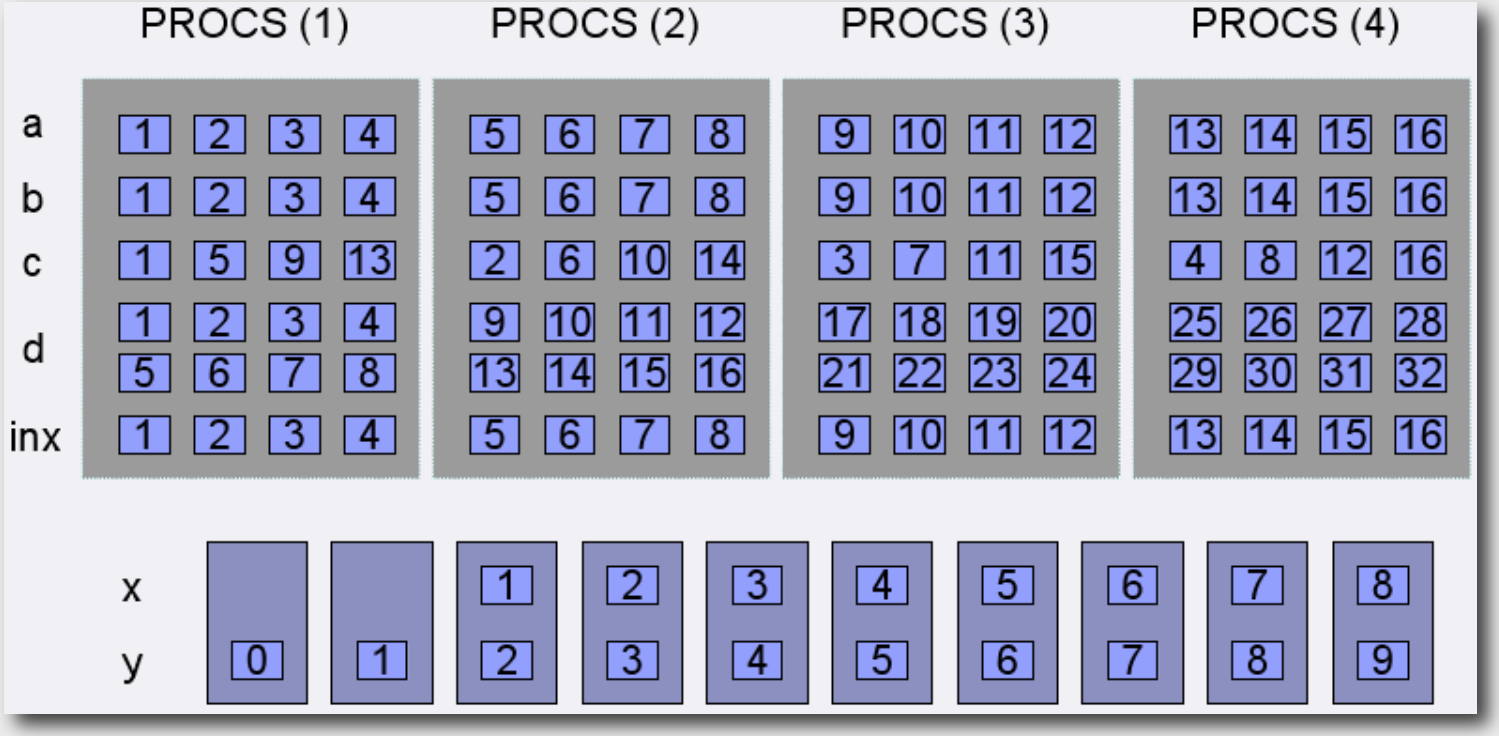


HPF: data mapping example


```
REAL, DIMENSION (16)   :: A, B, C
REAL, DIMENSION (32)   :: D
REAL, DIMENSION (8)     :: X
REAL, DIMENSION (0:9)  :: Y
INTEGER, DIMENSION (16) :: INX
!HPF$ PROCESSORS, DIMENSION(4)         :: PROC
!HPF$ DISTRIBUTE, (BLOCK) ONTO PROCS   :: A, B, D, INX
!HPF$ DISTRIBUTE, (CYCLIC) ONTO PROCS :: C
!HPF$ ALIGN X(I) WITH Y(I+1)
```

HPF: data mapping example

- the code in the previous slide is resulting as shown here:



HPF: Data distribution

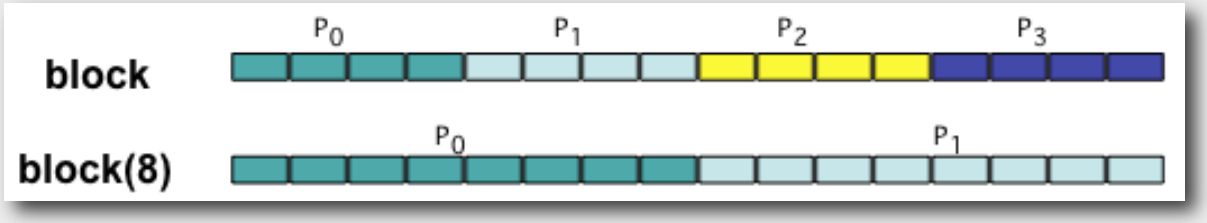
- Distributes the array elements over virtual processors
- Various predefined distribution functions
  - *block, block(m)*
  - *cyclic, cyclic(m)*
  - ** (all elements in dimension to same processor)*
- examples:

```
!HPF$ PROCESSORS, DIMENSION(4,4) ONTO p
!HPF$ PROCESSORS, DIMENSION(16) ONTO q
!HPF$ DISTRIBUTE a(block,block) ONTO p
!HPF$ DISTRIBUTE a(cyclic,*) ONTO q
!HPF$ DISTRIBUTE a(*,*) ONTO powerpc
```

HPF: block distribution

- BLOCK means give equal sized contiguous blocks of array elements to processors
- Suited for problems with equal computational load per element
- Example:

```
REAL a(16)
!HPF$        PROCESSORS p(4)
!HPF$        DISTRIBUTE a(block) ONTO p
or
!HPF$        DISTRIBUTE a(block(8)) ONTO
p
```



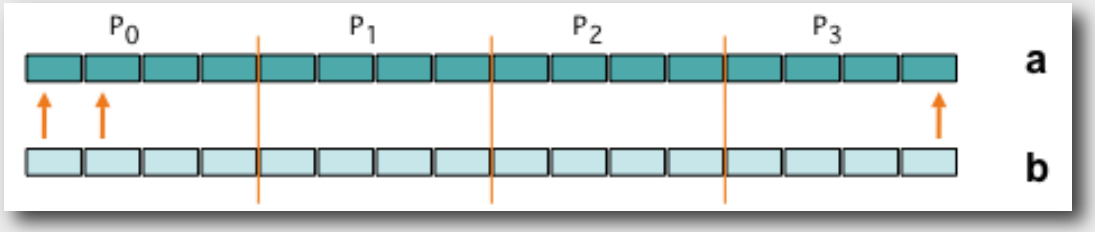
HPF: alignment

- Arrays can be ALIGNed to each other
- Aligned arrays are collectively distributed through top array
- Top array can be a TEMPLATE (no data, only shape)
- Purpose:
  - *Specify locality: two aligned array elements reside on the same processor*
  - *Specify replication and collapsing of dimensions*
  - *distribute a group of arrays with a single DISTRIBUTE statement*

HPF alignment example (1)

- Example: one-to-one alignment

```
REAL a(16), b(16)
!HPF$ PROCESSORS p(4)
!HPF$     ALIGN b(:) WITH a(:)
!HPF$     DISTRIBUTE a(block) ONTO
p
```



HPF alignment example (2)

- Example: collapsed alignment

```
REAL a(16), b(16,3)
!HPF$ PROCESSORS p(4)
!HPF$     ALIGN b(:,*) WITH a(:)
!HPF$     DISTRIBUTE a(block) ONTO p
```

HPF: parallelism

- F90 array assignment statements
- FORALL statement
- Examples:

array statements

```
REAL, DIMENSION(N)::a,b,c
a= b+c
a(1:10:2)=b(6:10) + c(1:5)
```

FORALL statement

```
FORALL (i=1:n,j=1:m, a(i,j).NE.0)
     a(i,j) = 1.0 / a(i,j)
```

HPF example: MVP


```
REAL a(m,n),y(m),x(n)
!HPF$         PROCESSORS p(4)
!HPF$         ALIGN a(:,*) WITH y(:)
!HPF$         ALIGN x(:) WITH y(:)
!HPF$         DISTRIBUTE y(block) ONTO p

!HPF$ INDEPENDENT
   DO j=1,n
     FORALL(i=1:m)
                        y(i) =y(i) + a(i,j)*x(j)
     END FORALL
   END DO
```

HPF example: MVP


```
REAL a(m,n),y(m),x(n)
!HPF$        PROCESSORS p(4)
!HPF$        ALIGN a(:,*) WITH y(:)
!HPF$        ALIGN x(:) WITH y(:)
!HPF$        DISTRIBUTE y(block) ONTO p

FORALL(i=1,m)
     y(i) = DOT_PRODUCT(a(i,:),x(:))
```

HPF: Summary

- Data-parallel programming model
  - Computation follows data
  - Communication and synchronization “by request”
- Data distribution & alignment
  - Essential for performance, locality, low communication/synchronization
- Different processors view
  - Allow for easier data distributions
- High abstraction level
- Performance depends on compiler
  - Large overheads depending on the machine

HPF Quiz:

Which of the following *conceptual programming models* be represented in HPF:

- *farmer/workers*
- *divide and conquer*
- *data parallelism*
- *task parallelism*
- *bulk synchronous*

- Write a short (pseudo-code) example for each one of those for which this is possible.
- Indicate the functionality
- Syntax correctness is not an issue.

Classical models:
Message Passing (MPI)

Distributed memory: message passing (MPI)

- Control:
  - *Independent processes*
  - *Typical SPMD*
    - *MPMD by process ID*
- Data:
  - *No notion of global data*
  - *Every process has local copies of the data*
- Communication
  - *By message passing*
  - *Variants: 1-1, 1-n, n-1, n-n*
- No special synchronization constructs
  - *implicit synchronization by message transfer*
  - *barrier synchronization is done by all-to-all broadcast*
- Abstraction level: low/medium

Message Passing: MVP Example


```
for(i=0; i<dim; i++) {
    y[i]=0;
    for(j=0; j<dim; j++) {
     y[i]=y[i] + A[i,j] * x[j]; } }
```

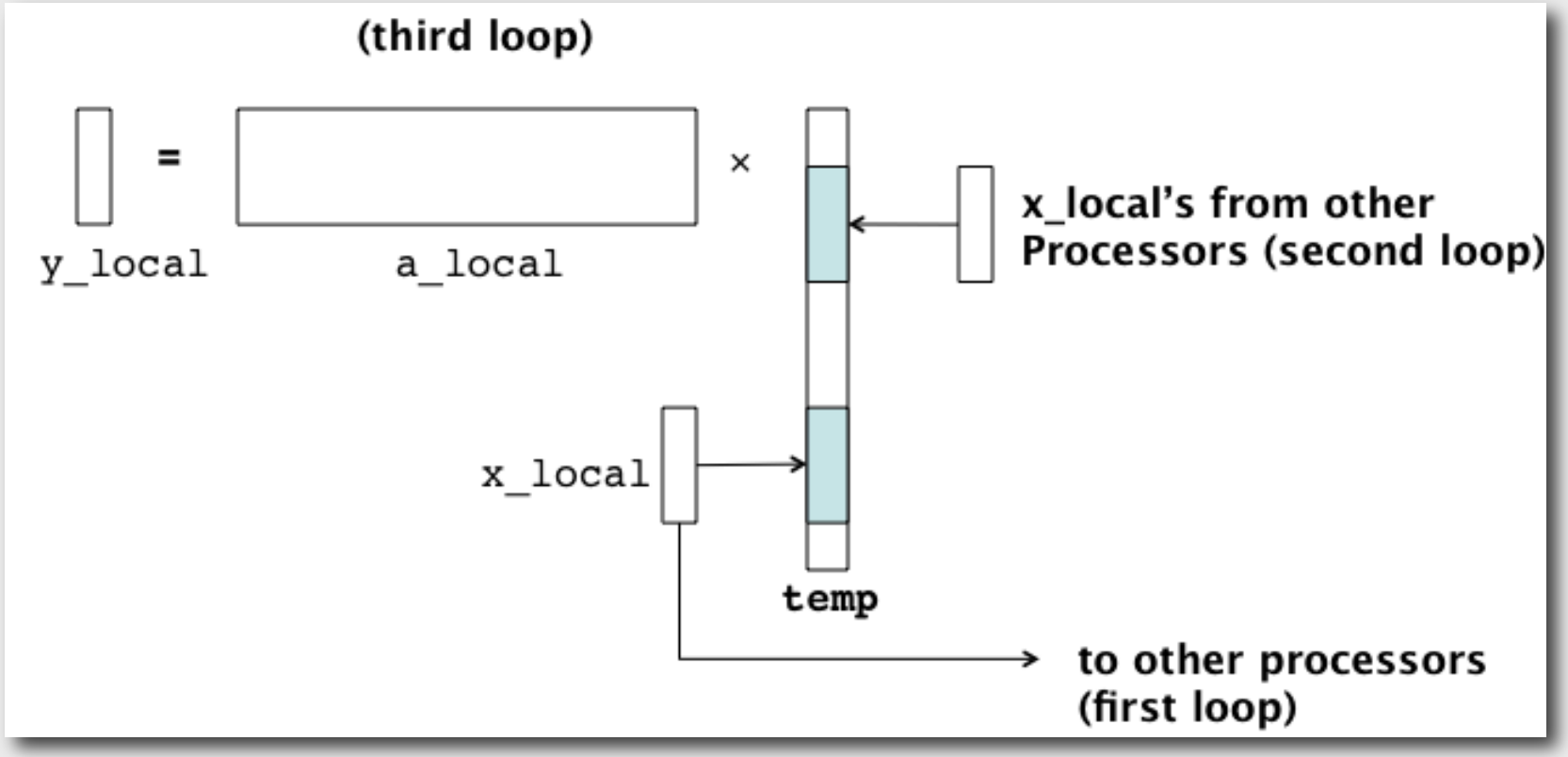
- Which processors gets which data?
  - A: chunk of rows, j= id * size .. (id +1)*size-1
  - x: chunk of elements, same j
- Communication?
- Synchronization?

Message Passing: MVP ver.01


```
float a_local[m/procs,n], y_local[m/procs], x_local[n/procs];
float temp[n];
int mypid;

mypid=proc(); k=n/procs;
for(i=0;i<procs;i++){
      if(mypid==i) send(&x_local[0],k,i);}
for(i=0;i<procs;i++){
      if(mypid!=i)
         receive(&temp[i*k],k,i);
     else
         copy(temp[i*k],k,x_local)
}
for(i=0;i<k;i++)
  for(j=0;j<n;j++)
   y_local[i]=y_local[i]+a_local[i,j]*temp[j];
```

MPV: data structures – ver.01



Message Passing: MVP – ver.01

- Memory
  - *Need local buffer (`temp[ ]`) of global size n anyway !*
- Performance issues?
  - *Communication with (proc-1) nodes*
    - *Sending n/proc items*
    - *Receiving n/proc items*
  - *Ordering of messages?*

Message Passing: MPV – ver 02

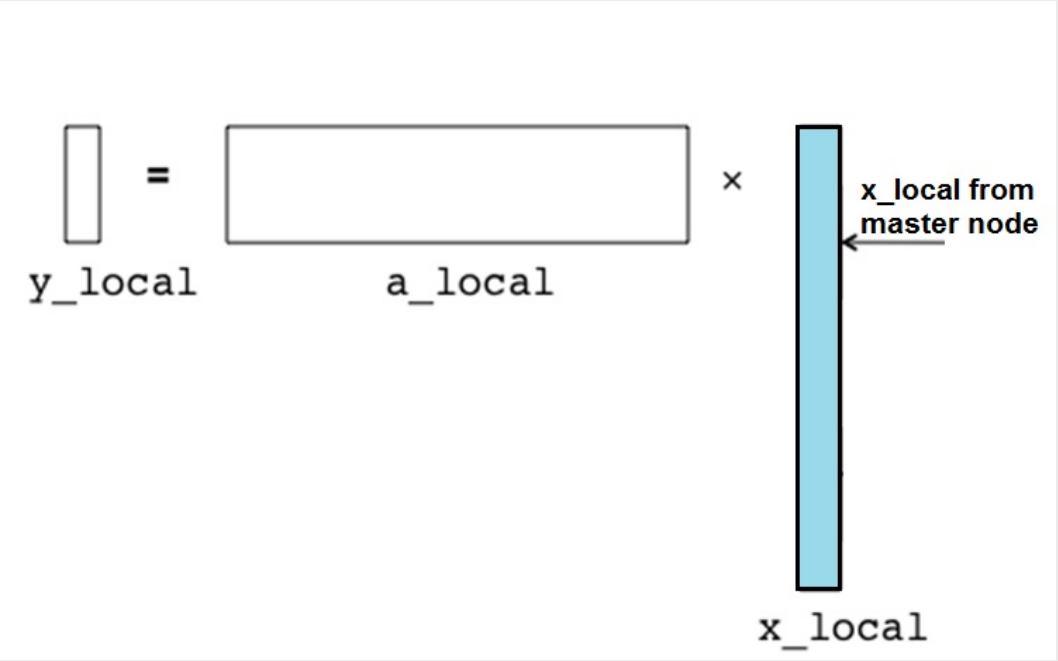
- another version of local structures results in:

```
float a_local[m/procs,n], y_local[m/procs],
x_local[n];
int mypid;

mypid=proc(); k=n/procs;
if (mypid==0)
     send_all(&x_local[0],n);
else
     receive(&x_local[0],n,0);

for(i=0;i<k;i++){
   for(j=0;j<n;j++){
     y_local[i]=y_local[i]+a_local[i,j]*x_local[j];
}}
```

MPV: data structures - ver.02



Message Passing: MVP – ver.02

- Memory
  - *No need for x_local[n/procs]*
  - *replace temp with full x_local*
- Performance issues?
  - *Broadcast : sending n items*
  - *Receiving n items*
  - *Ordering of messages?*

Message Passing: Summary

- No notion of global data
- Data communication is done by message passing
  - Expensive, performance-wise
- Trade-off between:
  - One-copy data
    - *More communication is needed, less consistency issues*
  - Local data replication
    - *(apparently) Less commnication, consistency is problematic*
- Techniques to improve performance:
  - Replicate read-only data
  - Computation and communication overlapping
  - Message aggregation

Message Passing Quiz

Which of the following *conceptual programming models* be represented in message passing (MPI):

- *farmer/workers*
- *divide and conquer*
- *data parallelism*
- *task parallelism*
- *bulk synchronous*

- Write a short (pseudo-code) example for each one of those for which this is possible.
- Indicate the functionality
- Syntax correctness is not an issue.

Next lecture

Next time – Lecture 4

- Parallel Programming Models (cont'd)
- Performance Analysis
  - Programming single-node
    - *Single CPU, Multi-core CPUs, GPUs*
  - Single node performance
  - Multi-node performance
- Reading assignment (optional)
  - *M.D.Hill et. al - Amdhal's Law in the Multi-core Era (July '08)*
  - *V.W.Lee et. al - Debunking the 100x GPU vs CPU Myth: an evaluation of throughput computing on CPU and GPU (June '10)*