High Performance Computing for parallel systems (051)

Michel Valin, Luc Corbeil

Environnement Canada

Outline

- Introduction
  - Basics of parallel Computing
  - Shared memory parallelism
    - Threads (high level, i.e. subroutine level)
    - OpenMP (low level, i.e. loop level)
  - Distributed memory parallelism
    - MPI (basic)
    - Toolkit (higher level operations)

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/

Outline (contd)

- Threads (basic concepts)
- OpenMP (basic concepts)
- MPI
  - basic concepts
  - Usage at CMC (compile, load, execute)
- RPN_COMM toolkit
  - Basic concepts
  - Usage at CMC (compile, load, execute)
- Try your luck !!

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/

Basics of Parallel Computing

- Why parallel computing ?
- Limits of a single processor
  - Speed
  - Real estate
- Workaround : parallelism
- Taxonomy of parallel computing
  - Hardware
  - Software
  - Memory

Why parallel computing ?

- You like new challenges
- It exists therefore it must be used
- The program will be real complicated, your job is secure
- It looks better in your resume
- ....
- You tried using only one processor and it still was not fast enough in real time. The guru helped you optimize it to the hilt and it still was not fast enough.

Single processor limitations

- Processor clock speed is limited
  - Physical size of processor limits speed because signal speed cannot exceed speed of light
  - Single processor speed is limited by integrated circuits feature size (propagation delays and thermal problems)
- Memory performance is limited (especially latency)
- The amount of logic on a processor chip is limited by real estate considerations (die size / transistor size)

Workaround : parallel computing

- Increase parallelism within processor (multi operand functional units like vector units)
- Increase parallelism on chip (multiple processor cores on chip)
- Multi processor chip computers
- Multi computer systems using a communication network (latency and bandwidth considerations)

Types of Parallel Computing
hardware wise

- Flynn's taxonomy (hardware oriented)
  - SISD (**S**ingle **I**nstruction **S**ingle **D**ata)
  - SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata)
  - MISD (**M**ultiple **I**nstruction **S**ingle **D**ata)
  - MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata)

Types of Parallel Computing
hardware wise
SISD

- SISD is an acronym for Single Instruction stream over a Single Data stream. It is a computing term referring to an architecture in which a single processor executes a single instruction stream, to operate on data stored in a single memory. Corresponds to the von Neumann architecture.

Types of Parallel Computing
hardware wise
SIMD

- SIMD (Single Instruction, Multiple Data) is a set of operations for efficiently handling large quantities of data in parallel. The first use of SIMD instructions was in vector supercomputers and was especially popularized by Cray in the 1970s.
- More recently, small-scale (64 or 128 bits) SIMD has become popular on general-purpose CPUs. SIMD instructions can be found to one degree or another on most CPUs, including the PowerPC's AltiVec, Intel's MMX, SSE, SSE2 and SSE3, AMD's 3DNow!. The instruction sets generally include a full set of vector instructions, including multiply, shuffle and invert.

Types of Parallel Computing

hardware wise

MISD

- Multiple Instruction Single Data (MISD) is a type of parallel computing architecture where many functional units perform different operations on the same data. Pipeline architectures (IBM CELL) belong to this type, though a purist might say that the data is different after processing by each stage in the pipeline. Not many instantiations of this architecture exist, as MIMD and SIMD are often more appropriate for common data parallel techniques. Specifically, they allow better scaling and use of computational resources than MISD does.
- Perhaps the only known practical application of MISD is for fault detection through redundant computation. Devices that need to achieve extremely high levels of reliability may implement two or more separate computational processes and check the results for consistency to ensure that all components are working correctly.

Types of Parallel Computing
hardware wise
MIMD

- Multiple Instruction Multiple Data (MIMD) is a type of parallel computing architecture where many functional units perform different operations on different data. Examples would be a multiprocessor computer, or a network of workstations.

Types of Parallel computing
software wise

- A programmer's taxonomy
  - Data-parallel : Same operation, different data
  - SPMD : **S**ingle **P**rogram **M**ultiple **D**ata
    - Same program, different data
  - MPMD : **M**ultiple **P**rogram **M**ultiple **D**ata
    - Different programs, different data
- (MPMD can be coerced to SPMD)

Types of parallel computing
memory wise

- The memory model
  - SMP Shared Memory Parallelism
    - One processor can “ peek into ” another processor's memory
    - Cray X-MP, single node NEC SX-3/4/5/6, IBM pSeries
  - DMP Distributed Memory Parallelism
    - Processors exchange “ messages ”
    - Cray T3D, IBM SP, ES-40, ASCI machines

SISD

- Scalar processors
- 1 CPU
  - Bendix G20
  - IBM 360
  - CDC 7600
  - CRAY 1 / NEC SX / Fujitsu VPP scalar instructions
  - Personal computer (not too recent !!)
  - IBM Power 4/5 series processors

SIMD

- Parallelism with **single** control
- Several functional units
- **One** control unit
- Examples:
  - Illiac IV
  - CRAY 1 (and other single CPU vector processors)
  - Thinking machines CM-2
  - Intel SSE
  - Altivec instruction set (PowerPC)

MIMD

- Free running parallelism
- Several **INDEPENDENT** control units
    - Each with several functional units
- Examples:
    - CRAY X-MP
    - NEC SX-3/4/5/6 (one node)
    - SGI Challenge/Origin
    - HP K200/K400 series
    - SMT (Simultaneous Multi Threading)(Intel , IBM Power)

SPMD

- ONE program
- SEVERAL sets of data
  - Shared memory
    - Threads
    - Multitasking (thread oriented)
    - OpenMP (loop oriented) (also known as Microtasking)
  - Distributed memory
    - MPI (process oriented)
    - PVM (process oriented)

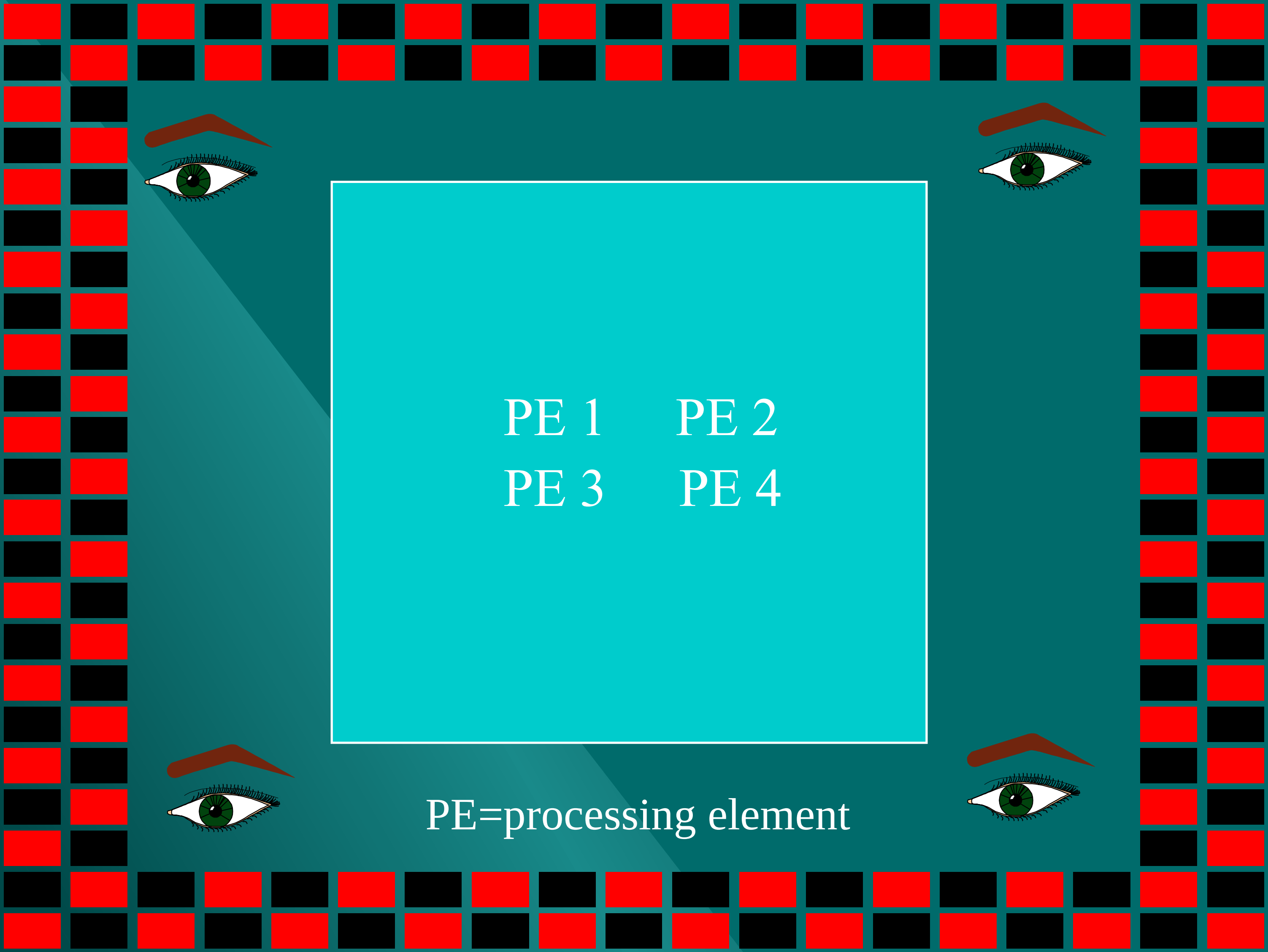
MPMD

- SEVERAL programs (processes)
- SEVERAL sets of data
  - Distributed memory
    - MPI   [+OpenMP]
    - PVM   [+OpenMP]
- Can be coerced to SPMD if necessary in the case of a par-allel application (a name space merge operation must be performed in this case)
- Trivial case: multiprogramming environment on a multi-user computer system (this cannot be coerced to SPMD !!)

SMP
Shared Memory Parallelism

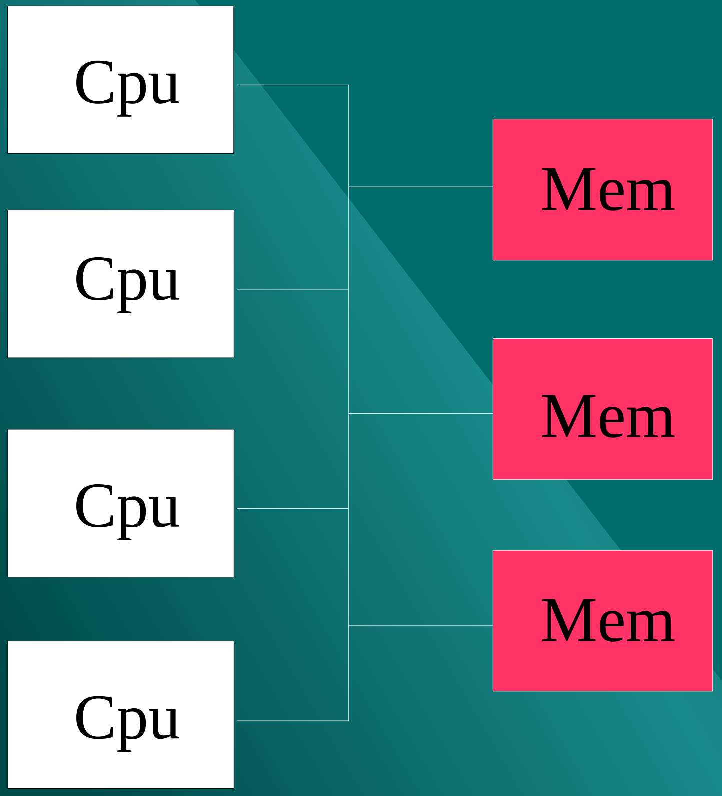
- Easier programming paradigm
- SPMD or MPMD
- Main points to pay attention to
  - Load splitting
  - Load balancing
- Limitations
  - Amdahl's law (single threaded portion)
  - Memory bandwidth (shared memory access path)
- Multithreading on a multi-cpu machine is a classical case

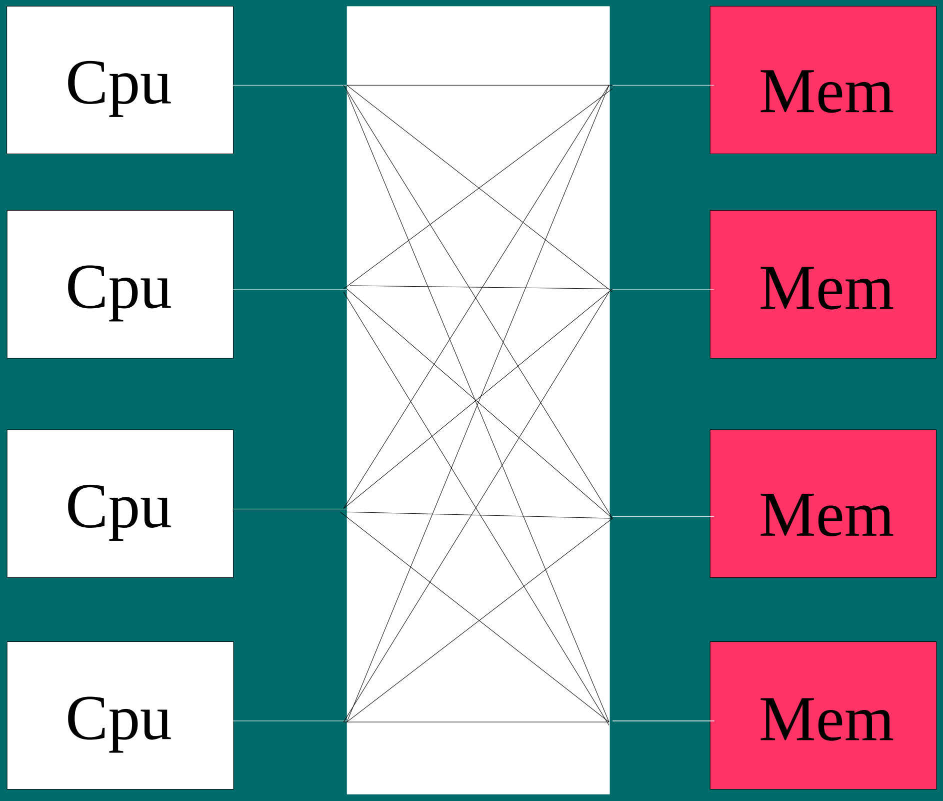
SMP
Shared Memory Parallelism



PE=processing element

SMP architectures



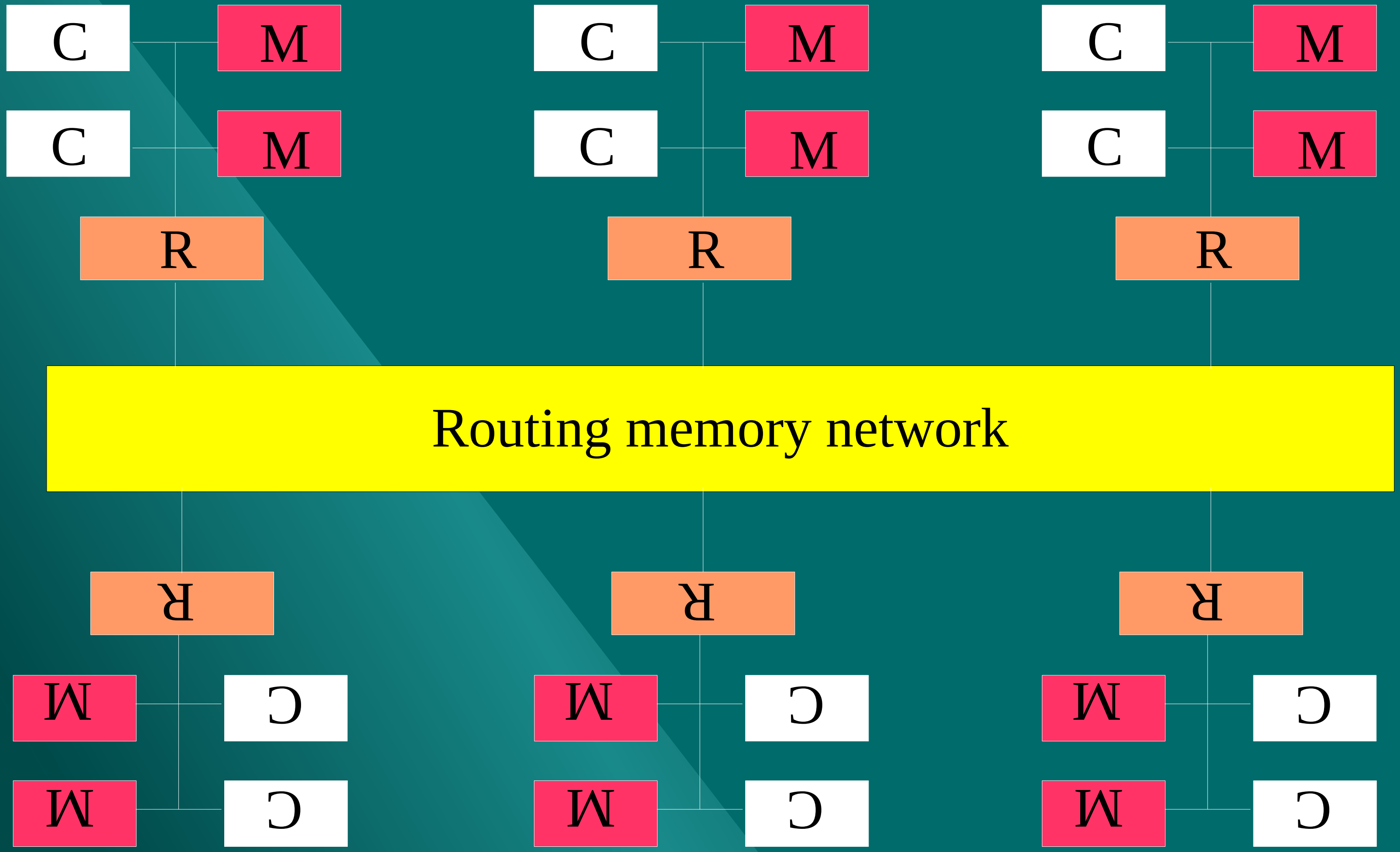


Bus topology

Network / crossbar

Uniform **M**emory **A**ccess

SMP architectures (contd)



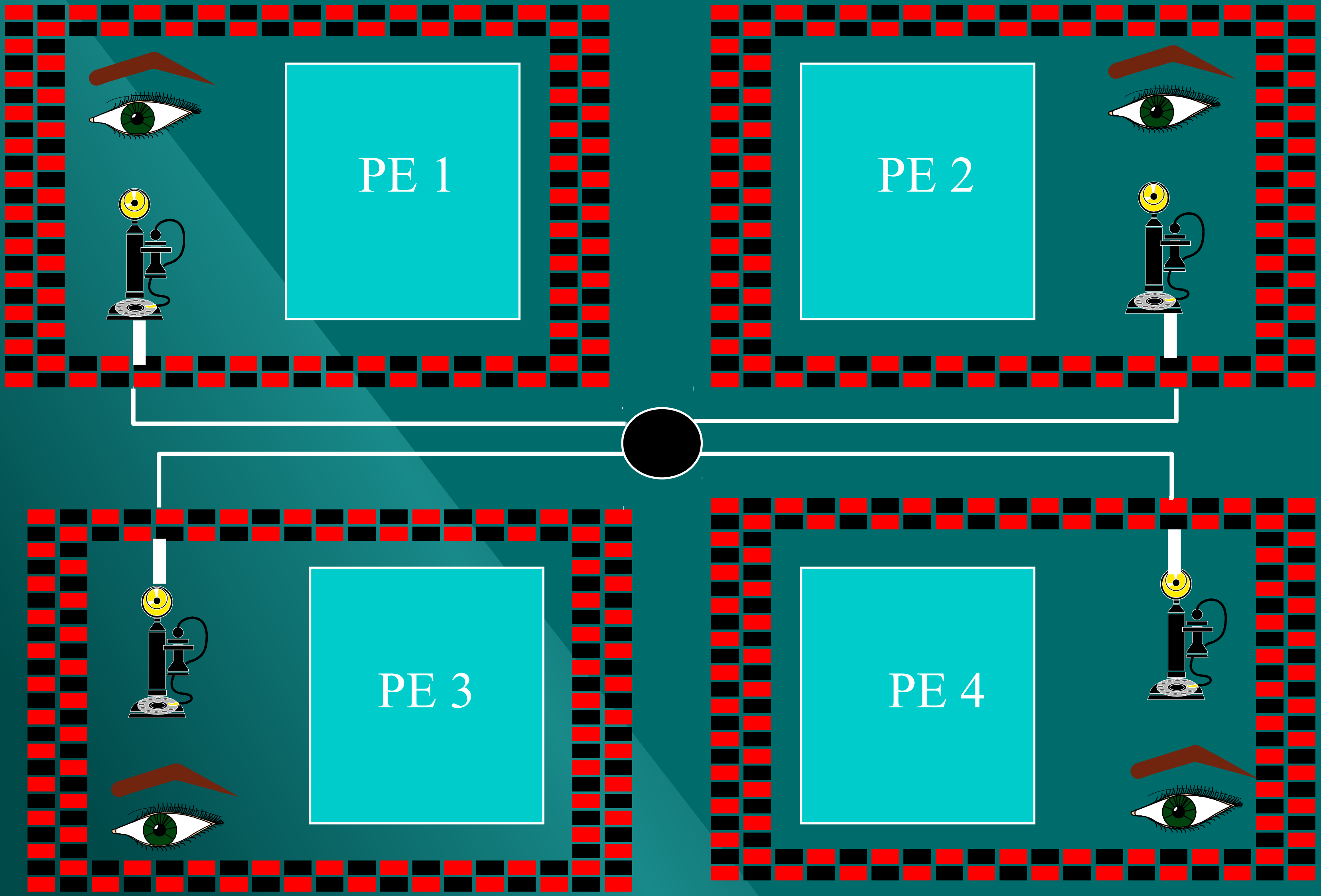
Non**U**niform**M**emory**A**ccess

DMP
Distributed Memory Parallelism

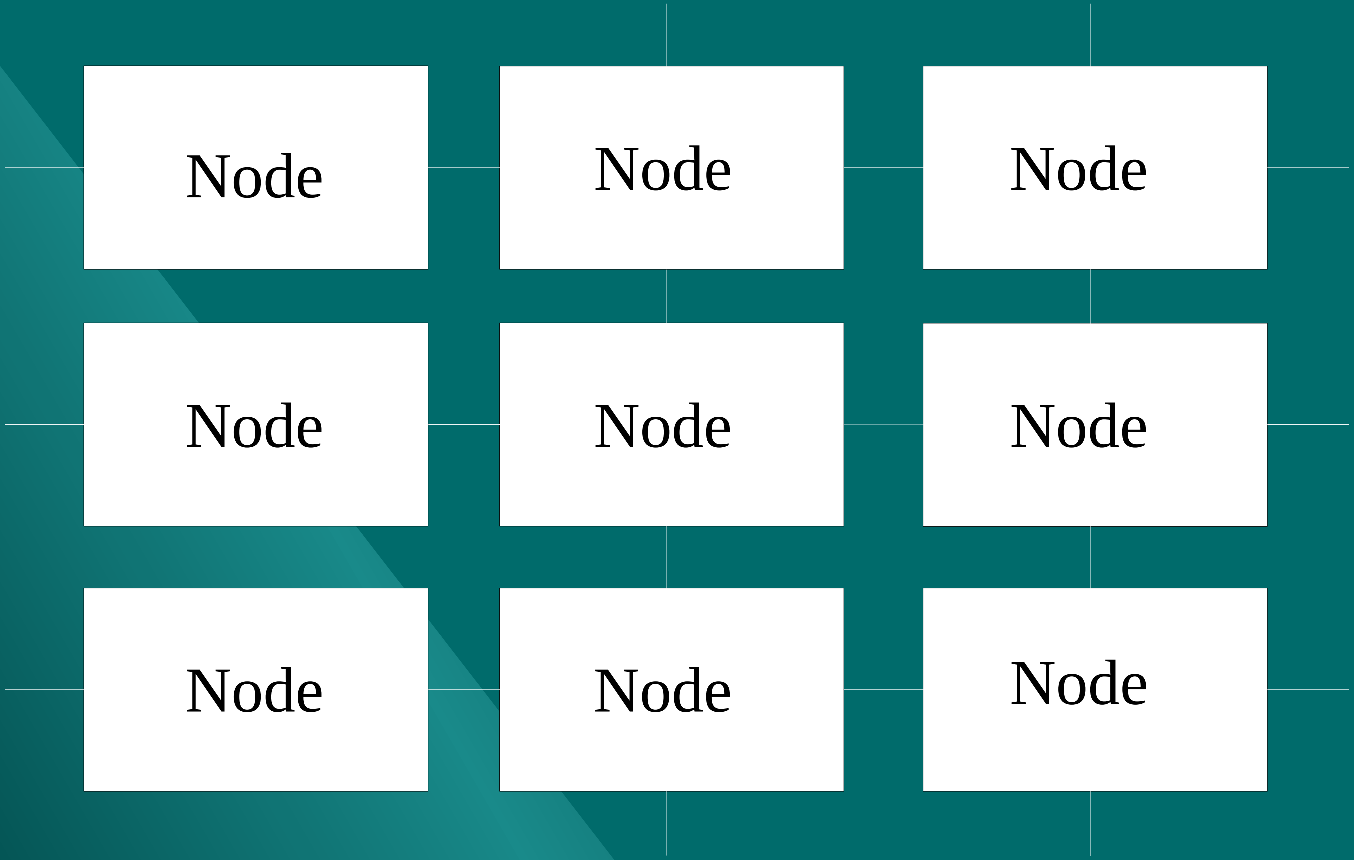
- More difficult but more powerful programming paradigm
- SPMD or MPMD
- Main points to pay attention to
  - Load splitting
  - Load balancing
  - Domain decomposition
- Limitations
  - Amdahl's law (duplicated execution or non parallel)
  - Communication bandwidth and latency

DMP
Distributed Memory Parallelism

PE=processing element

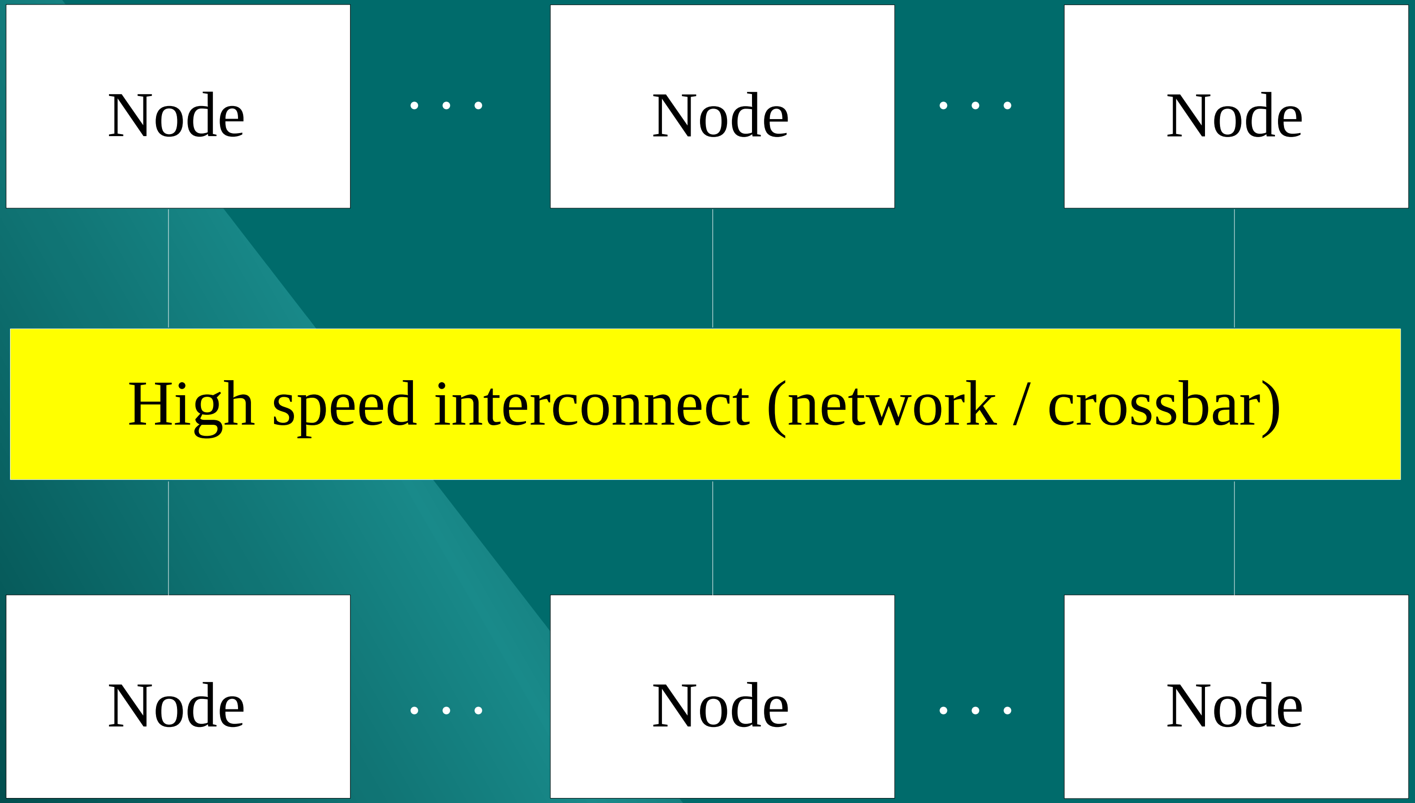


Distributed memory architecture



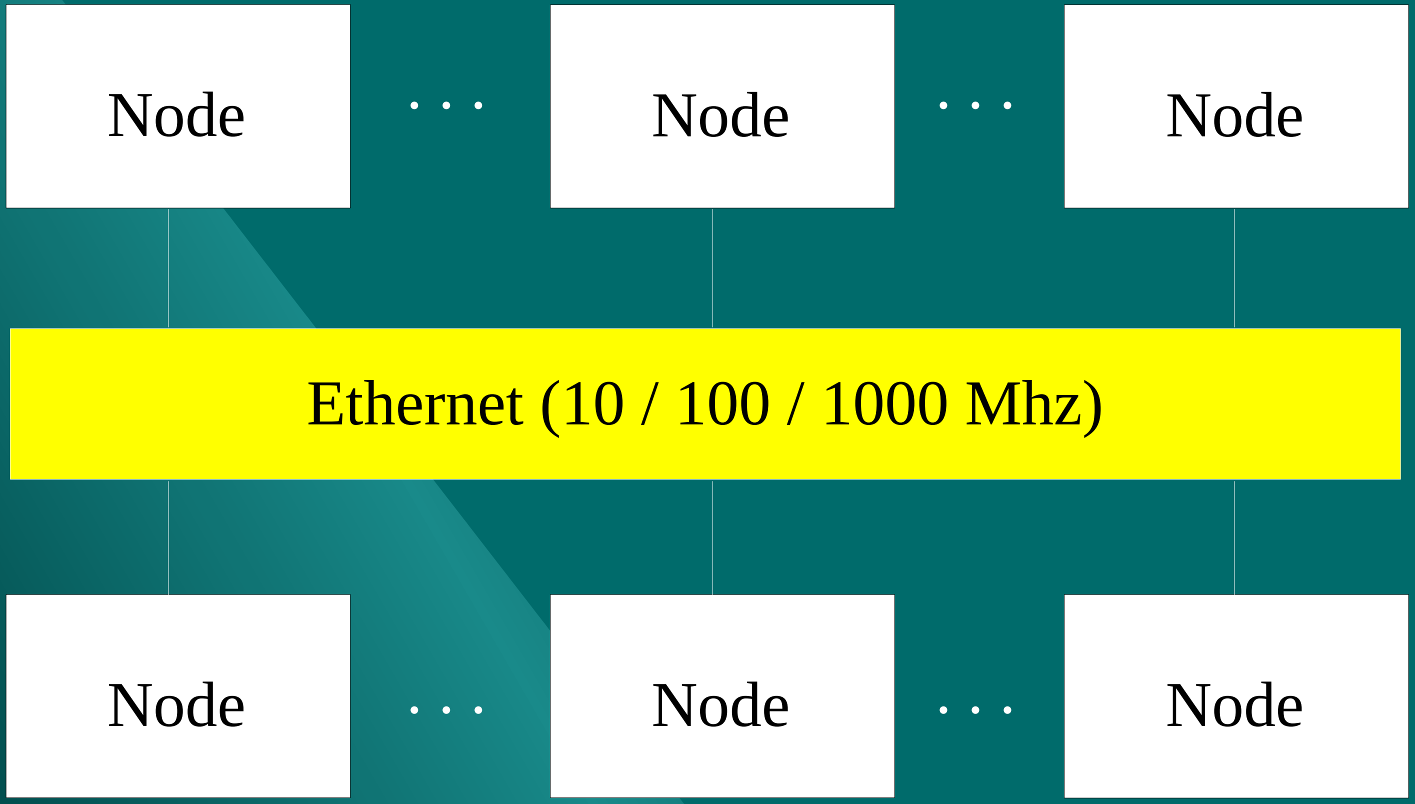
Lattice (2D/3D grid / torus)
(CRAY T3, XT3)

Distributed memory architecture



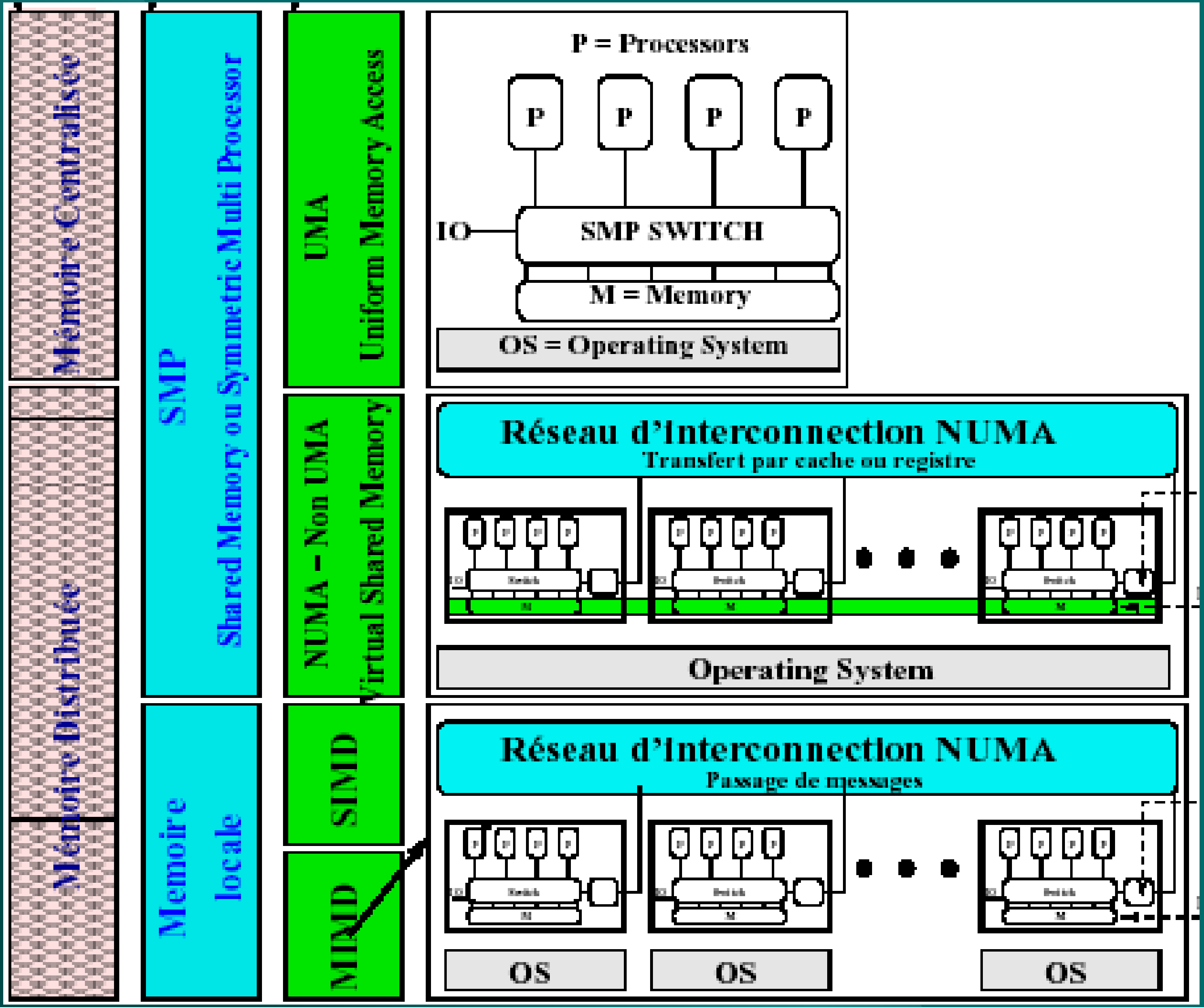
Network (InfiniBand, Quadrics, IXS,
Scalable Coherent Interconnect,...)

Distributed memory architecture



Network

Summary



NUMA controllers

Global address space

Single OS image

Network controllers

Local address spaces

Multiple OS images

Amdahl's law for parallel processing

- Given an amount of work W = Ws + Wp
  - Ws = serial work (cannot be divided)
  - Wp = parallel work (can be divided)
- If R is the processing speed
- The execution time will be
  - T1cpu = Ws/R + Wp/**R**
  - Tncpu = Ws/R + Wp/**(n * R)**

Corollary

- The speedup factor will be
  - S = T1cpu / Tncpu
  - S = W / (Ws + Wp/n)
  - Efficiency = S / ncpu
- If ncpu = 10 and Ws/W = .1 (9% serial work)
  - S = 1 / (.1 + .9/10) = 5.26 (efficiency = 52.6% BOF !!)
- If ncpu = 100 and Ws/W = .1
  - S = 1 / (.1 + .9/100) = 9.17 (efficiency = 9.17% YUK !!!)

The bottom line

- The speedup factor is influenced very much by the residual serial (non parallelizable) work. As the number of processors grows, so does the damage caused by non parallelizable work.

Distributed memory supercomputers

- Scalar uniprocessors
  - CRAY T3D/T3E
- Scalar Multiprocessors
  - IBM SP family (pSeries)
- Vector uniprocessors
  - FUJITSU VPP-5000
  - (HITACHI SR-8000)
- Vector multiprocessors
  - NEC SX-4/5/6/7
  - CRAY X1

Shared memory parallelism

- Threads
  - Basics
  - Examples
- OpenMP
  - Basics
  - Examples

Threads (multitasking)

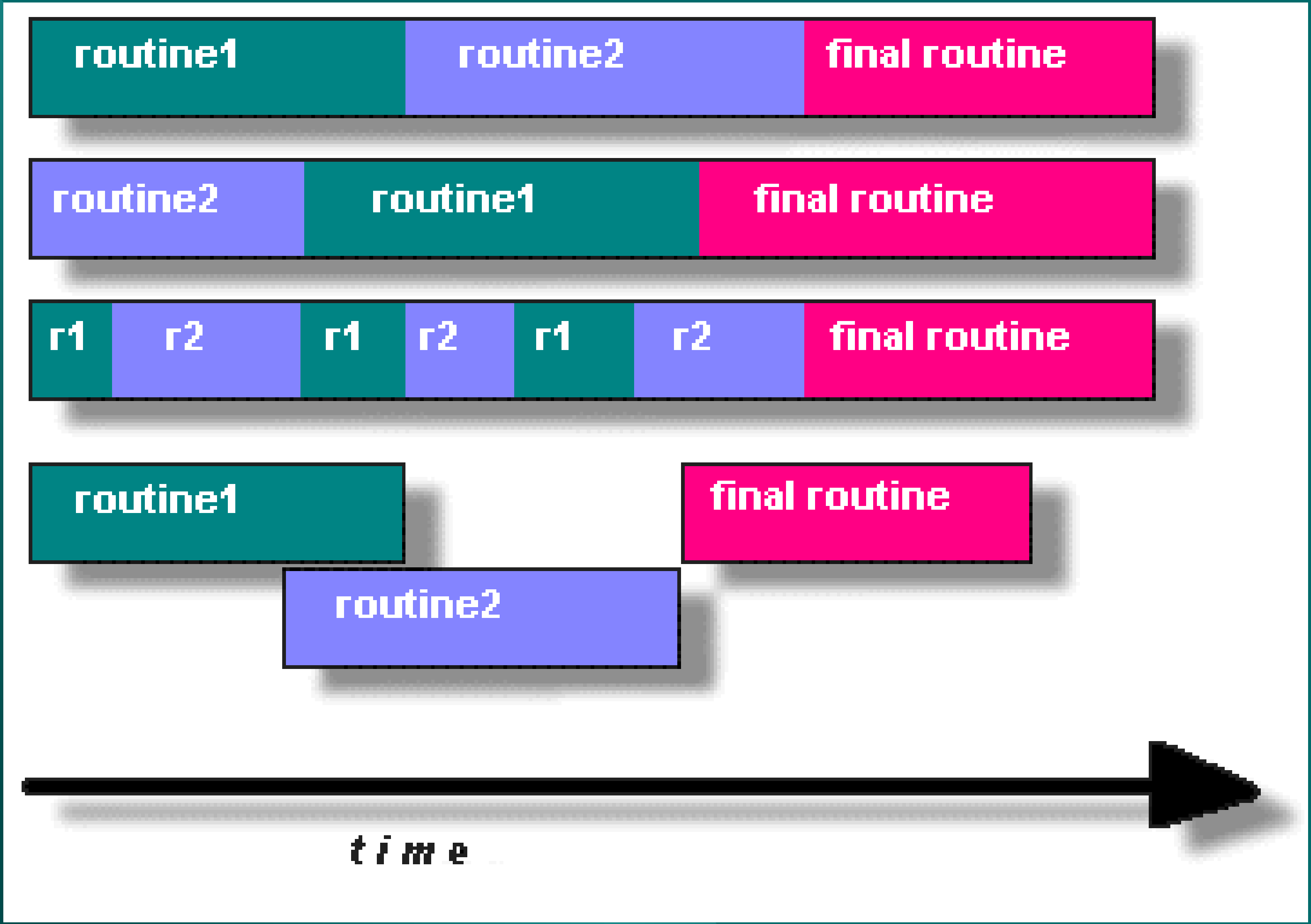
- Threads are light weight processes, oriented to-wards large to very large granularity, that share a common memory space.
- Basic functionality (subroutine calls)
  - Start a thread (usually a high level subroutine)
  - Wait for thread termination
  - Terminate thread forcibly
  - Manage events
  - Manage locks
- POSIX threads have a **C** API, but a FORTRAN one is relatively easy to code

Threads

- What are threads?
- A thread is a sequence of instructions to be executed within a program. Normal UNIX processes consist of a single thread of execution that starts in main(). In other words, each line of your code is executed in turn, exactly one line at a time. Before threads, the normal way to achieve multiple instruction sequences (ie, doing several things at once, in parallel) was to use the fork() and exec() system calls to create several processes -- each being a single thread of execution.

Threads

http://www.llnl.gov/computing/tutorials/pthreads/



IF is OK

And IF is OK

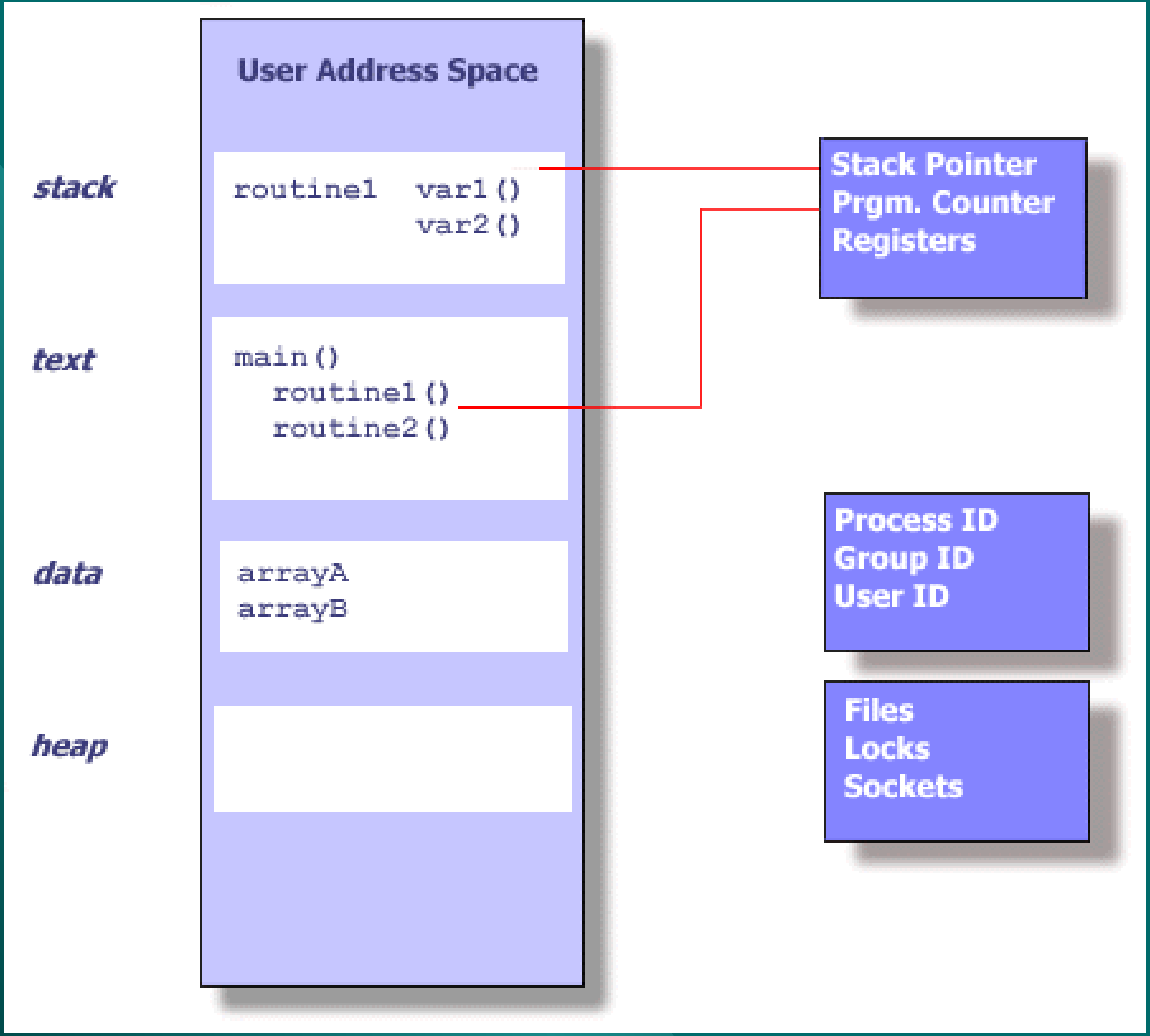
THEN

Threads

```
Time             Master Thread
 |                    |
 |                    |        create workers with pthread_create()
 |                    |
 |                  // \\      workers start up
\ /                /|   |\
 |                | |   | |
 |                | |   | |     workers do their jobs
 |                | |   | |
 |                 \|   |/
 |                  \\ //      workers terminate
 |                    |
 |                    |        join workers with pthread_join()
\ /                   |
 |              Master Thread
```

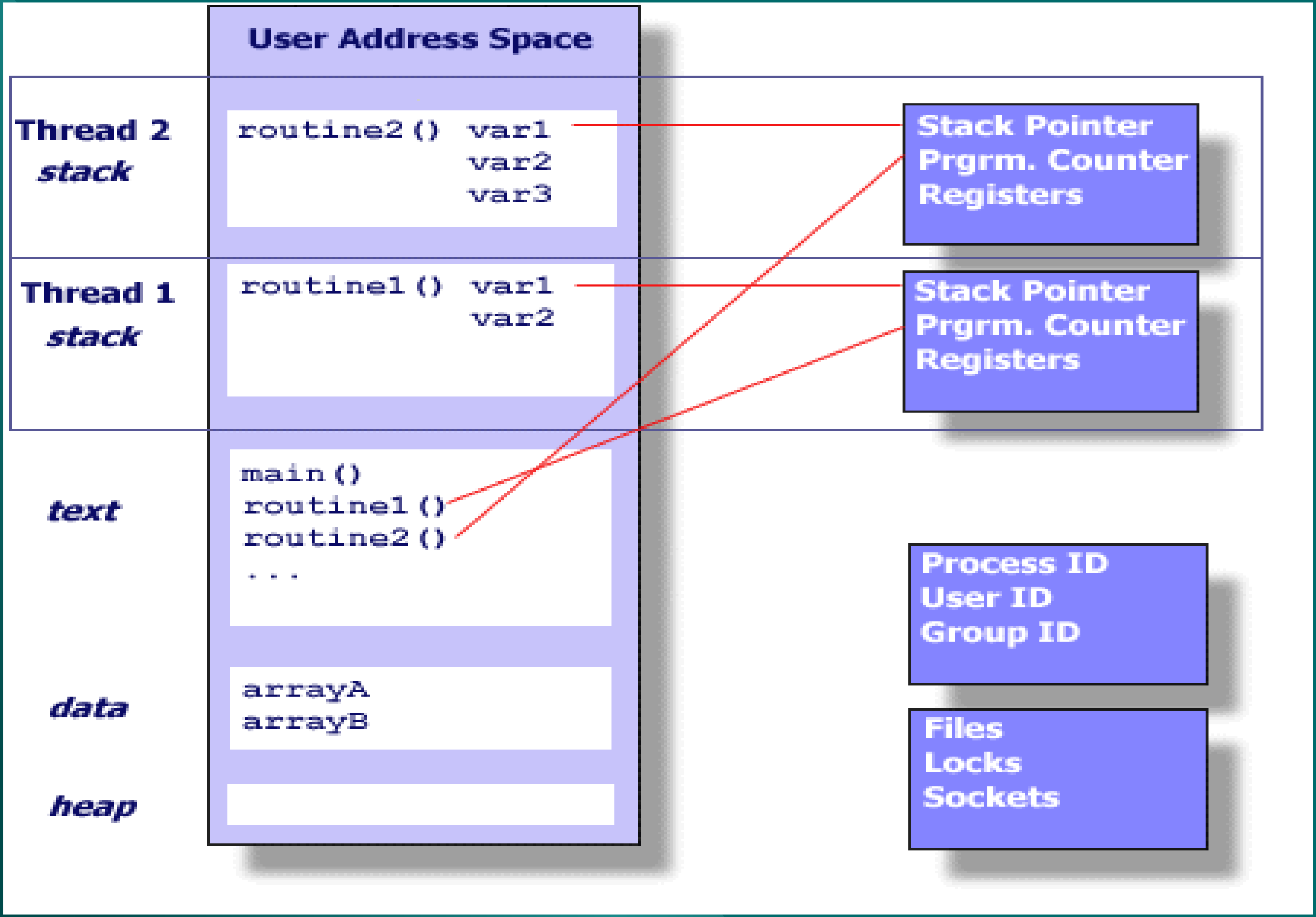
Threads (memory layout)

http://www.llnl.gov/computing/tutorials/pthreads/



Threads (memory layout)

http://www.llnl.gov/computing/tutorials/pthreads/

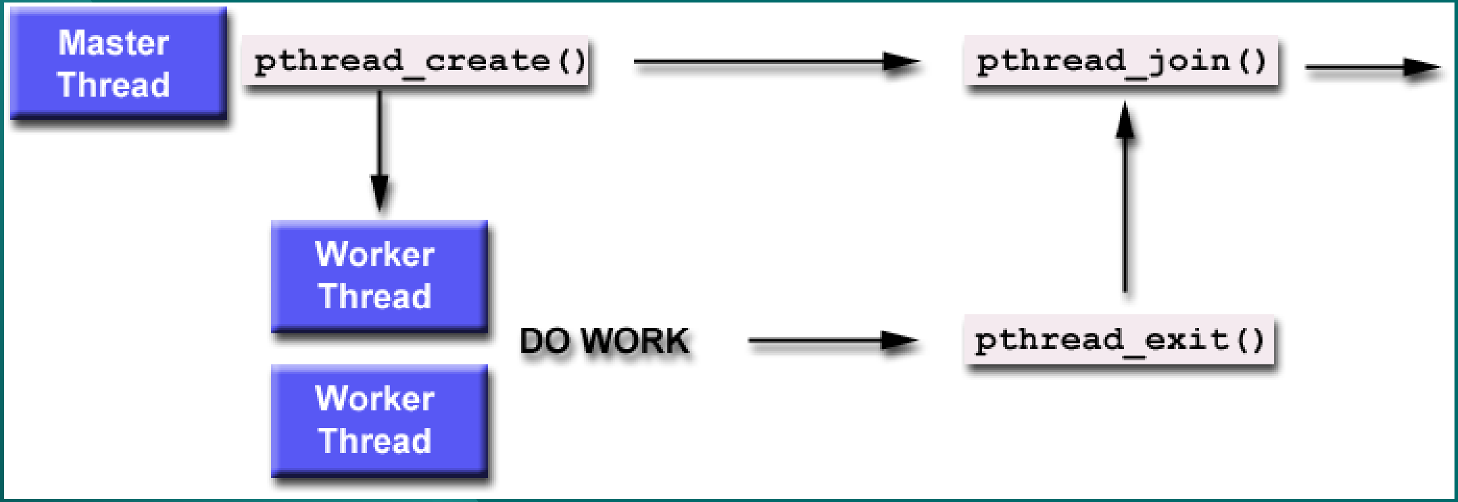


Threads (memory layout)

```
+-----------------------------------------------------------------------+
| Process                                                               |
|                                                                       |
|   +-------+     +-------------+  +-------------+  +-------------+     |
|   | Files |     | Thread      |  | Thread      |  | Thread      |     |
|   +-------+     |+-----------+|  |+-----------+|  |+-----------+|     |
|                 || Registers ||  || Registers ||  || Registers ||     |
|                 |+-----------+|  |+-----------+|  |+-----------+|     |
|                 |             |  |             |  |             |     |
| ....................................................................  |
| .               |             |  |             |  |             |  .  |
| . Memory        |             |  |             |  |             |  .  |
| .               | +---------+ |  | +---------+ |  | +---------+ |  .  |
| . +--------+    | |  Stack  | |  | |  Stack  | |  | |  Stack  | |  .  |
| . |  Heap  |    | |         | |  | |         | |  | |         | |  .  |
| . +--------+    | |         | |  | |         | |  | |         | |  .  |
| .               | |         | |  | |         | |  | |         | |  .  |
| . +--------+    | |         | |  | |         | |  | |         | |  .  |
| . |  Data  |    | |         | |  | |         | |  | |         | |  .  |
| . +--------+    | |         | |  | |         | |  | |         | |  .  |
| .               | |         | |  | |         | |  | |         | |  .  |
| . +--------+    | |         | |  | |         | |  | |         | |  .  |
| . |  Code  |    | +---------+ |  | +---------+ |  | +---------+ |  .  |
| . +--------+    +-------------+  +-------------+  +-------------+  .  |
| .                                                                  .  |
| ....................................................................  |
|                                                                       |
+-----------------------------------------------------------------------+
```

Threads

http://www.llnl.gov/computing/tutorials/pthreads/



Threads (basic C functions)
man pthreads for more information

- pthread_create (thread,attr,start_routine,arg)
- pthread_join (threadid,status)
- pthread_self ()
- pthread_mutex_init (mutex,attr)
- pthread_mutex_lock (mutex)
- pthread_mutex_unlock (mutex)
- pthread_cond_init (cond,attr)
- pthread_cond_broadcast (cond)
- pthread_cond_wait (cond,mutex)

Threads

http://www.llnl.gov/computing/tutorials/pthreads/

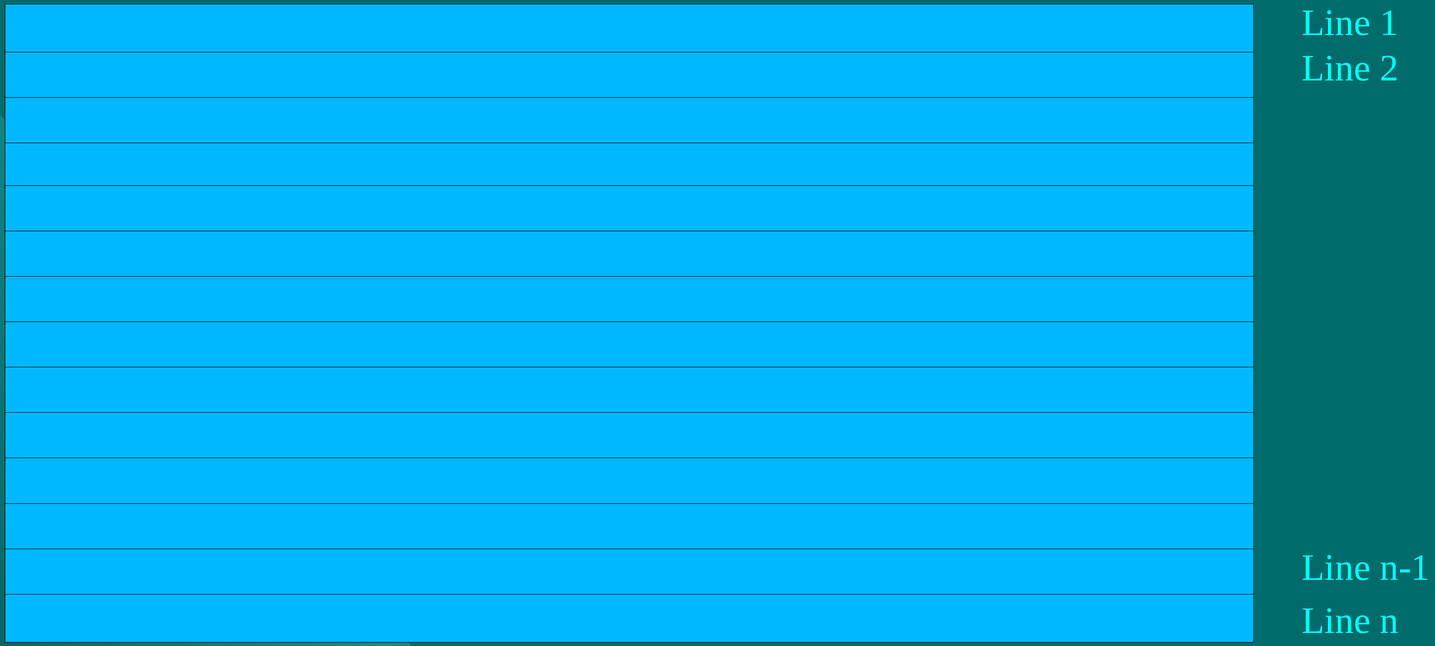
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS;t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

FORTRAN Thread functions

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/THREADS

- INTEGER*8   INTEGER   ANY_TYPE
- thread_id = CREATE_THREAD(function,arg)
- status = JOIN_THREAD(thread_id)
- thread_id = ID_THREAD()
- status = CREATE_LOCK(lock)
- status = ACQUIRE_LOCK(lock)
- status = RELEASE_LOCK(lock)
- status = CREATE_EVENT(event)
- status = POST_EVENT(event,value)
- value = CHECK_EVENT(event)

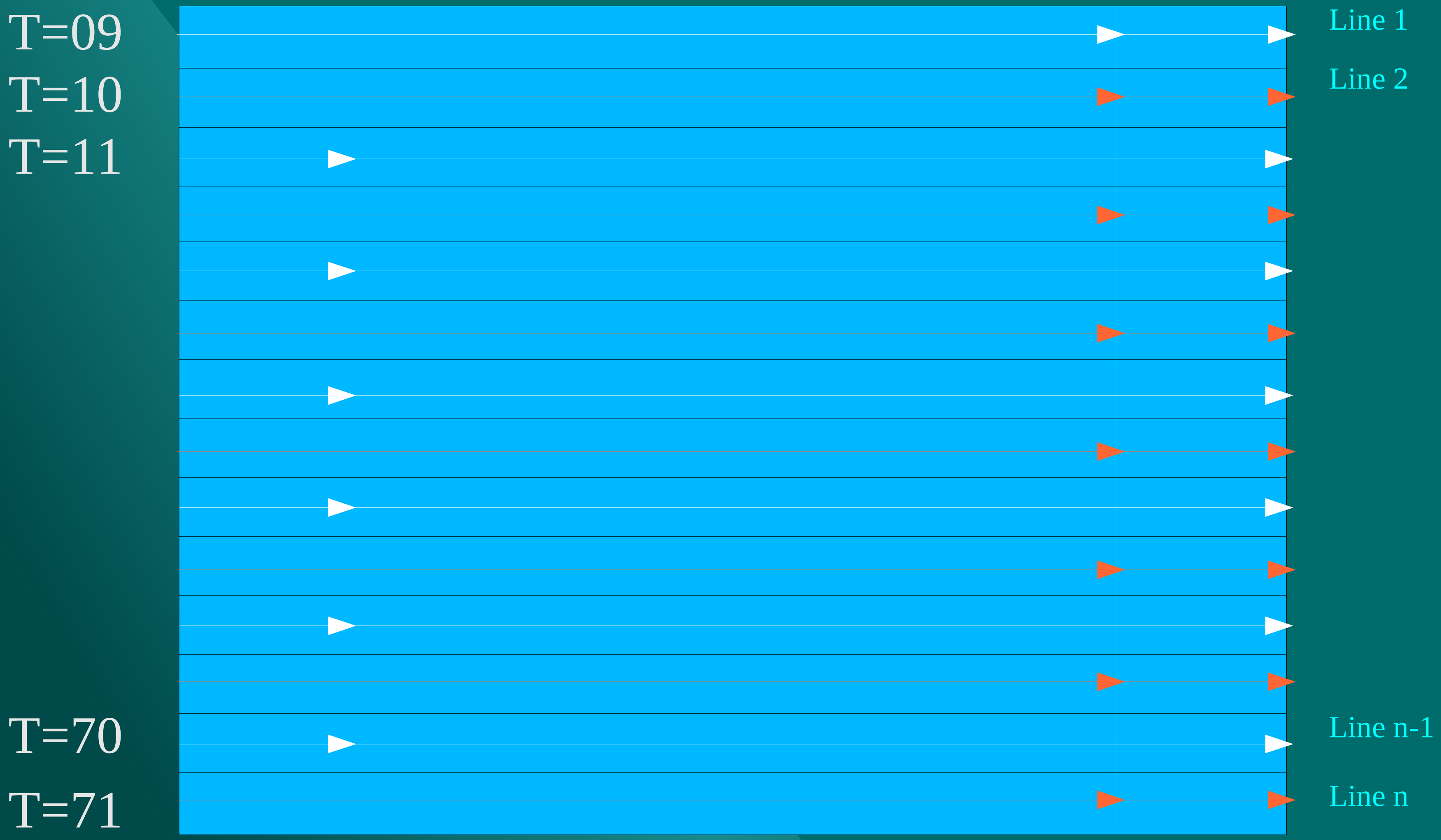
FORTRAN Thread functions



For each line in array :

1) perform some operations on line that are independent of other lines
2) perform some operations that need the results from the previous line

FORTRAN Thread functions



FORTRAN Thread example

```
program thread_demo
include 'demo_f90_threads.h'
integer *8 liste(10)
real array(10)
pointer(pvar,var)
integer id(10),create_thread
external proxy

icur=0          ! reset starting point counter
ntodo=50        ! 50 rows "to do"
do i=1,50       ! create locks and events
  irows(i)=0
  call create_lock(locks(i))
  call create_event(iready(i))
enddo
```

```
do i=1,5                    ! start worker threads
  array(i)=i
  write(6,*)'starting thread',i
  call flush(6)
  id(i)=create_thread(proxy,array(i))
  write(6,*)'START OF THREAD, ID=',id(i)
  call flush(6)
enddo
write(6,*)'THREADS CREATED'
call flush(6)
call proxy(0.0)             ! main thread

write(6,*)'done processing, joining'
call flush(6)

do i=1,5  ! wait for worker threads to terminate
  call join_thread(id(i))
enddo

print *,'icur=',icur,' ntodo=',ntodo
stop
end
```

FORTRAN Thread example

```
 subroutine proxy(element)  ! row processing code
 real element
 include 'demo_f90_threads.h'
 integer icurl,ntodol
 external do_something
 integer do_something

 print *,'start of thread ',id_thread(),' arg=',element
100     continue

 call acquire_lock(locks(1))     ! increment global index icur, copy into local index
 icur=icur+1
 icurl=icur
 call release_lock(locks(1))
 if(icurl.gt.50)goto 200          ! no work left, return

 print *,'Processing row ',icurl

 isum=0
 do ii=icurl,50                   ! check that nobody went beyond current position
   isum=isum+irows(ii)
 enddo
 if(isum .ne.0) print *,'READY ??!!'     ! this should NEVER happen
```

FORTRAN Thread example

```
isum=1000000   ! DO SOME "PRE WORK" on row icurl
do i=1,1000000
      isum=do_something(isum)
enddo

if(icurl .gt. 1) then            ! wait for row icurl-1 to be "ready"
   print *,'Waiting for row ',icurl-1
   call wait_event(iready(icurl-1),1)
endif

isum=800000  !  DO SOME "POST WORK" on row icurl
do i=1,10000
  isum=do_something(isum)
enddo
irows(icurl)=1                   ! post row icurl as being "ready"
if(icurl .le. 50) call post_event(iready(icurl),1)
print *,'row ',icurl,' is ready'

call acquire_lock(locks(2))      ! decrement global "to do" counter
ntodo=ntodo-1
ntodol=ntodo
call release_lock(locks(2))
if(ntodol .gt. 0)   goto 100        ! no work left, return
200     continue
return
```

OpenMP (microtasking / autotasking)

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/OpenMP

- The OpenMP application programming interface (API) supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- The core elements of OpenMP are the constructs for thread creation, work load distribution (work sharing), data environment management, thread synchronization.

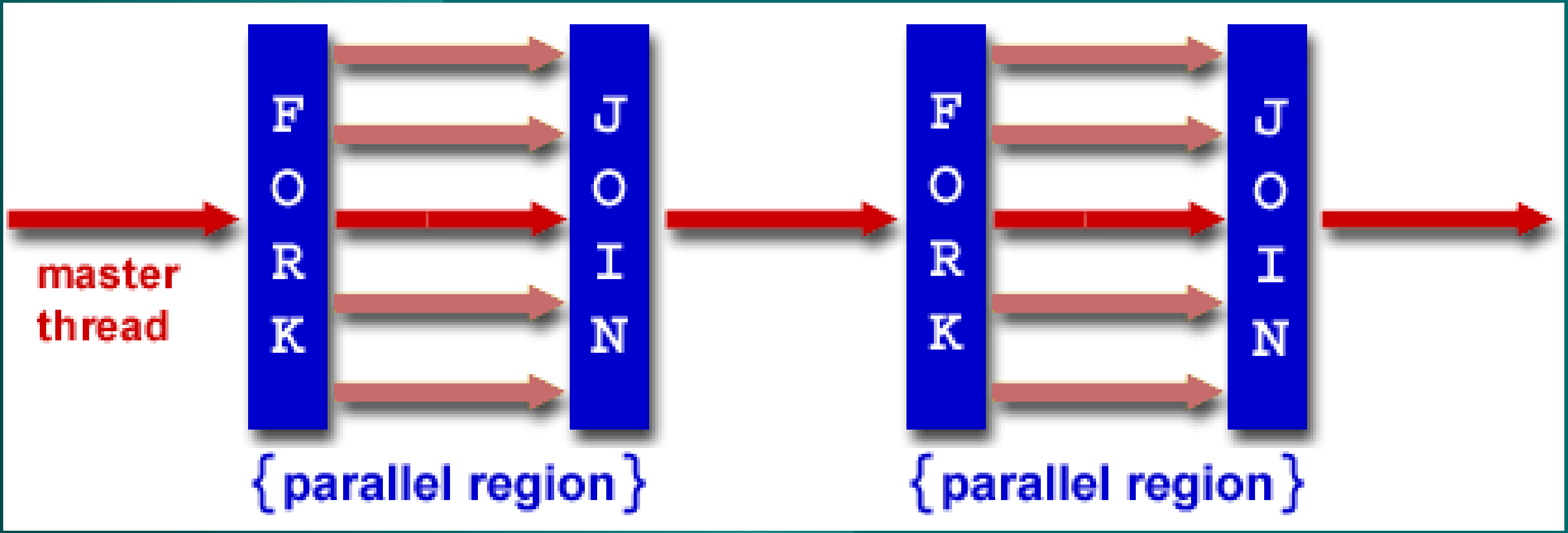
OpenMP (microtasking / autotasking)

- OpenMP works at the loop level (small granularity often at the loop level), multiple CPUs execute the same code in a shared memory space
- Basic features (FORTRAN “comments”)
  - Begin of parallel region
  - End of parallel region
  - Manage critical regions (**one** CPU at a time)
  - Manage serial regions (executed only **once**)
  - Variable scope management (shared vs private)

OpenMP

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. Compilers may offer options to generate directives automagically.

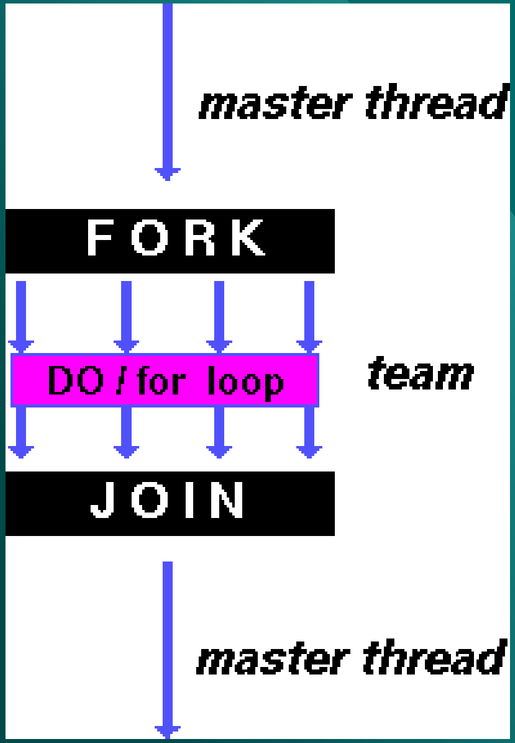
OpenMP uses the **fork-join** model of parallel execution:

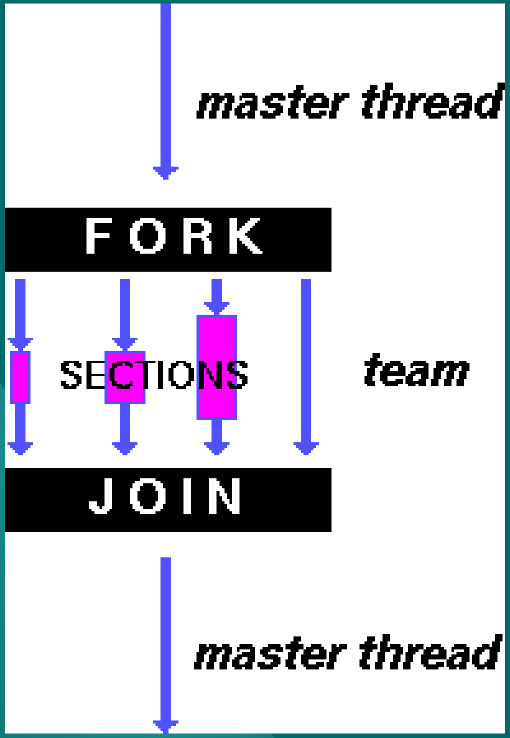


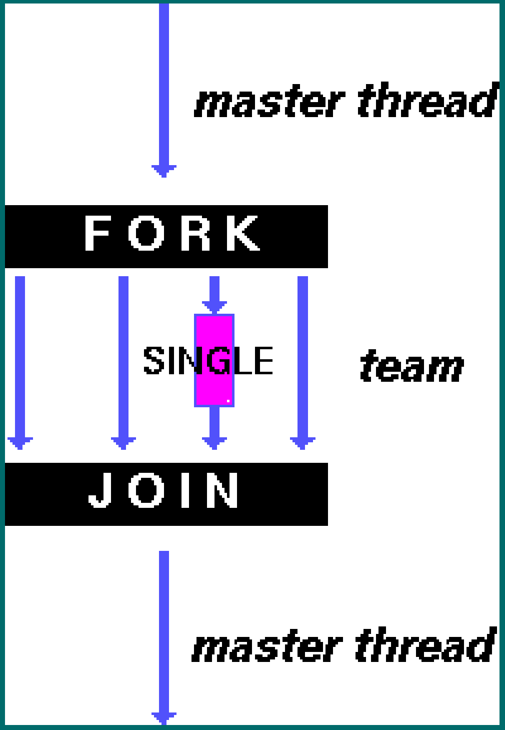
OpenMP

```
PROGRAM HELLO

       INTEGER VAR1, VAR2, VAR3

       Serial code

       Beginning of parallel section. Fork a team of threads.
       Specify variable scoping

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)

       Parallel section executed by all threads
             .
             .

       All threads join master thread and disband

!$OMP END PARALLEL

Resume serial code

       END
```

OpenMP


master thread
FORK
DO / for loop
team
JOIN
master thread


master thread
FORK
SECTIONS
team
JOIN
master thread


master thread
FORK
SINGLE
team
JOIN
master thread

OpenMP

```
PROGRAM VEC_ADD_DO

      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=1000)
      PARAMETER (CHUNKSIZE=100)
      REAL A(N), B(N), C(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO
      CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

END
```

OpenMP

```
PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO

!$OMP PARALLEL SHARED(A,B,C), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
      DO I = 1, N/2
         C(I) = A(I) + B(I)
      ENDDO
!$OMP SECTION
      DO I = 1+N/2, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL

END
```

OpenMP

```
PROGRAM CRITICAL

      INTEGER X
      X = 0

!$OMP PARALLEL SHARED(X)

!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL

!$OMP END PARALLEL

END
```

```
PROGRAM CRITICAL

      INTEGER X


!$OMP PARALLEL SHARED(X)

!$OMP MASTER
      X = 0
!$OMP END MASTER
!$OMP BARRIER

!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL

!$OMP END PARALLEL

END
```

```
PROGRAM CRITICAL

      INTEGER X


!$OMP PARALLEL SHARED(X)

!$OMP SINGLE
      X = 0
!$OMP END SINGLE
!$OMP BARRIER

!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL

!$OMP END PARALLEL

END
```

OpenMP Pros (cheap)

- Simple: no need to deal with message passing as MPI does
- Data layout and decomposition is handled automatically by directives.
- Incremental parallelism: can work on one portion of the program at one time, no dramatic change to code is needed.
- Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used. (possible need for some function stubbing)
- Original (serial) code statements needs not, in general, to be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.

OpenMP Cons (you get what you pay for)

- Currently only runs efficiently on shared-memory multiprocessor platforms
- Requires a compiler that supports OpenMP.
- Low parallel efficiency: relies more on parallelizable loops, potentially leaving out a relatively high percentage of a non-loop code in sequential part.

General remarks

- Shared memory parallelism at the loop level can often be implemented after the fact if what is desired is a moderate level of parallelism (and speedup)
- It can be also done to a lesser extent at the thread level in some cases but reentrancy, data scope (thread local vs global) and race conditions can be a problem.

  (you can lookup reentrant or thread-safe on wikipedia)

Distributed memory parallelism

- Basic concepts
  - Communication costs
  - Communication through messages
    - Cooperative
    - One sided
  - Data decomposition
  - High level data movement
- MPI
- RPN_COMM

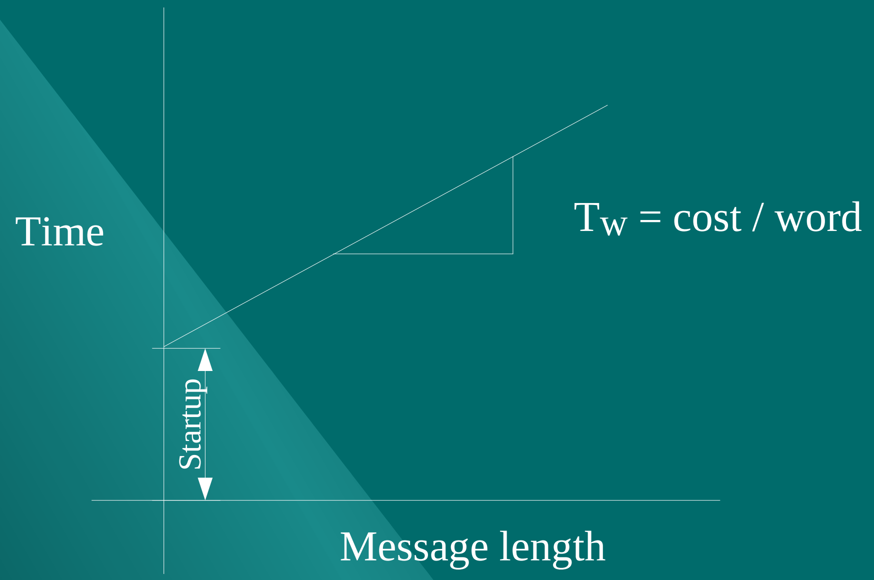
General remarks

- Distributed memory parallelism does not happen, it must be DESIGNED.
- One does not parallelize a code, the code must be rebuilt (and often redesigned) taking into account the constraints imposed upon the dataflow by message passing. Array dimensioning and loop indexing are likely to be VERY HEAVVILY IMPACTED.
- One may get lucky and HPF or an automatic parallelizing compiler will solve the problem [if one believes in (a) miracles, (b) Santa Claus, (c) the tooth fairy or (d) all of above].

Messages

- If memory is not shared between processors (distributed), the only way to communicate information from one part of the system to another is through the use of messages. A message is a data packet sent from one processor (sender) to another processor (receiver) in an organized fashion (just like the post office)
- Communications through messages can be
  - Cooperative send / receive (democratic)
  - One sided get / put (autocratic)

Communication costs

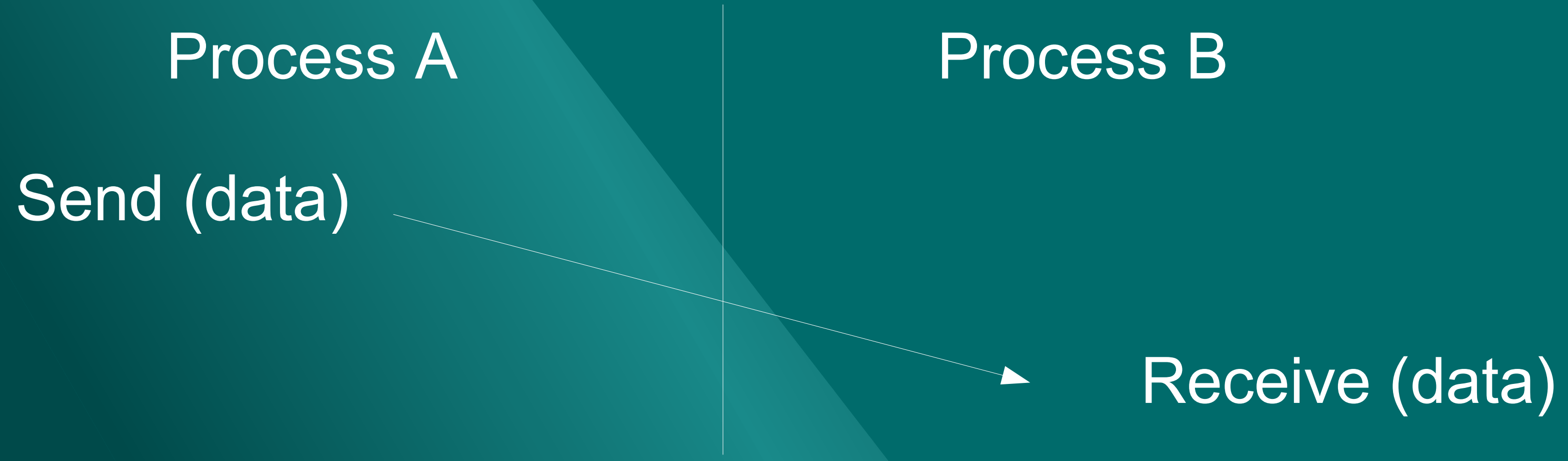


Communication costs examples

Machine	latency (microseconds)	data transfer rate (MegaBytes/second)
IBM Power 4	25	290
IBM Power 5	5	1700
Intel Paragon	120	60
CM-5	82	8
Ncube-2	150	2
GiGEthernet	40-120	100
Infiniband	5-30	160-900
NEC SX6	5	10 000

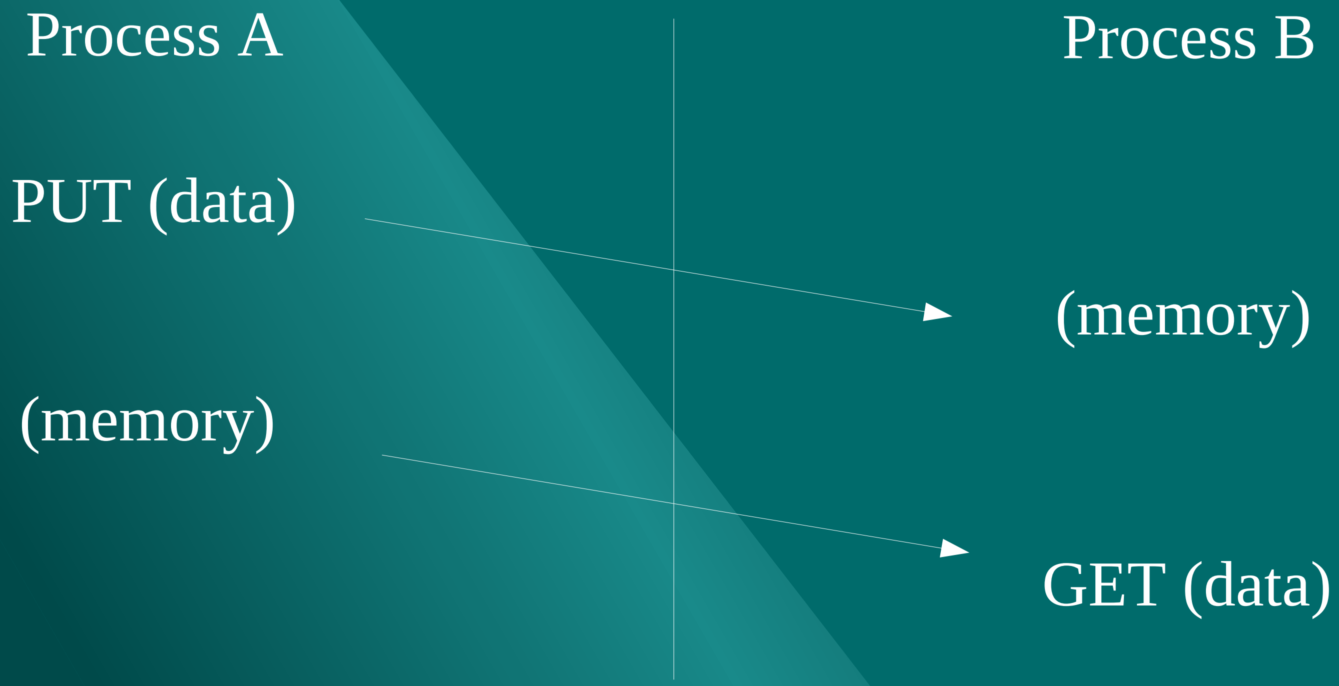
Cooperative communications

- The exchange of data is handled through message passing
- Data is EXPLICITELY sent and received
- Advantage: any change in receiver's memory is made with receiver's participation, all participants know what is going on and when it is going on.



One sided communications

- Remote memory reads and writes (hardware assisted)
- Pros: data can be accessed without waiting for other process
- Cons: synchronization may not be that easy



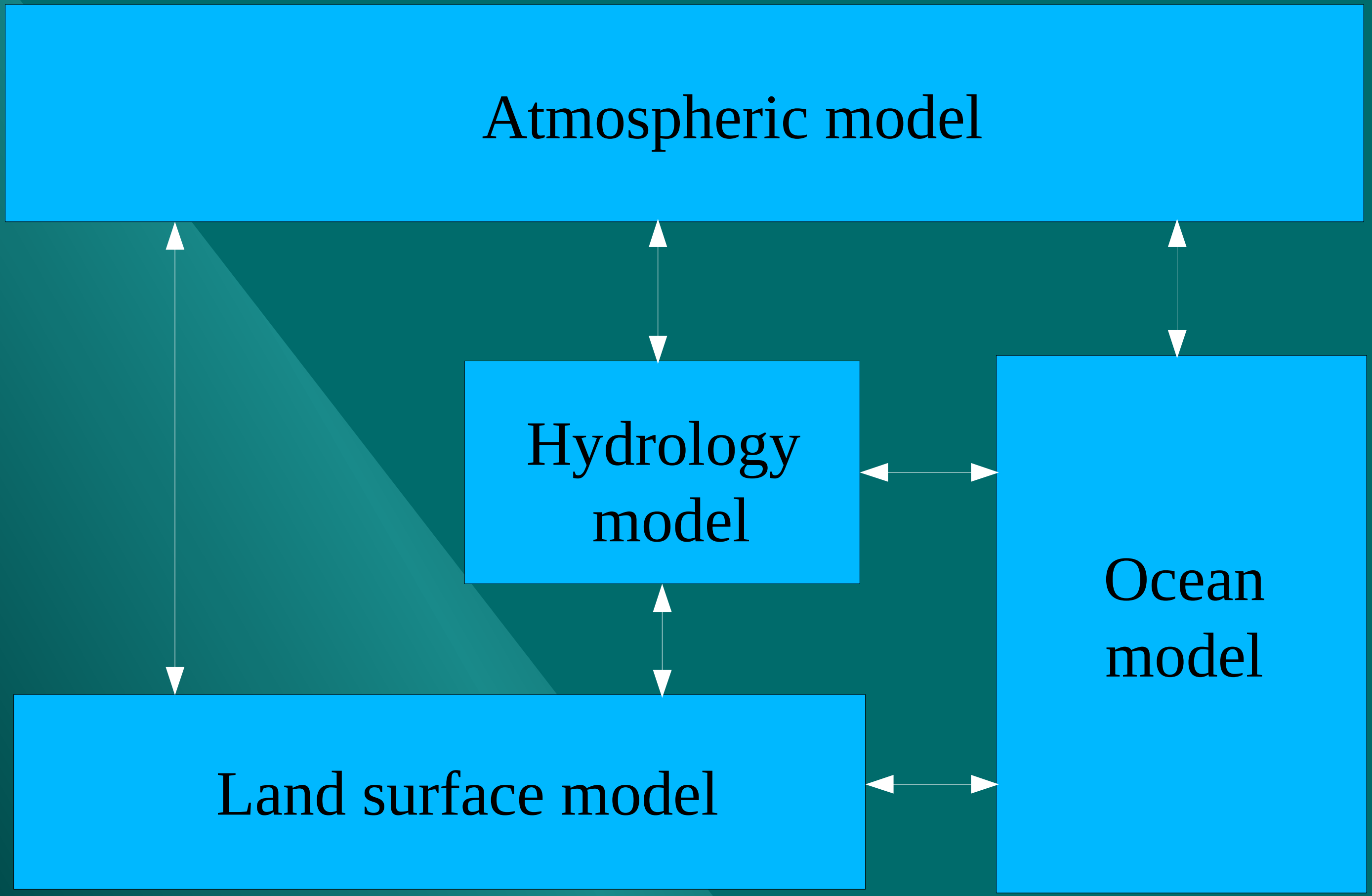
Decomposition

- Basic checklist
- Functional decomposition
- Domain decomposition
  - Global coordinates (points)
  - Local coordinates (points and processors)
- 1 D decomposition
- 2 D decomposition

Basic partitioning checklist

- Does your partitioning define many more tasks than there are cpus in target computer ? If not, little flexibility.
- Does your partitioning avoid redundant computations and storage ? If not, algorithm may not scale well.
- Are tasks of comparable size ? If not, load balance suffers.
- Does number of tasks scale with problem size ? If not, it may be difficult to use more processors for larger problem
- Have you identified alternative partitioning ? It is best to consider alternatives at beginning. Look at domain and functional decomposition (or a mix).

Functional decomposition

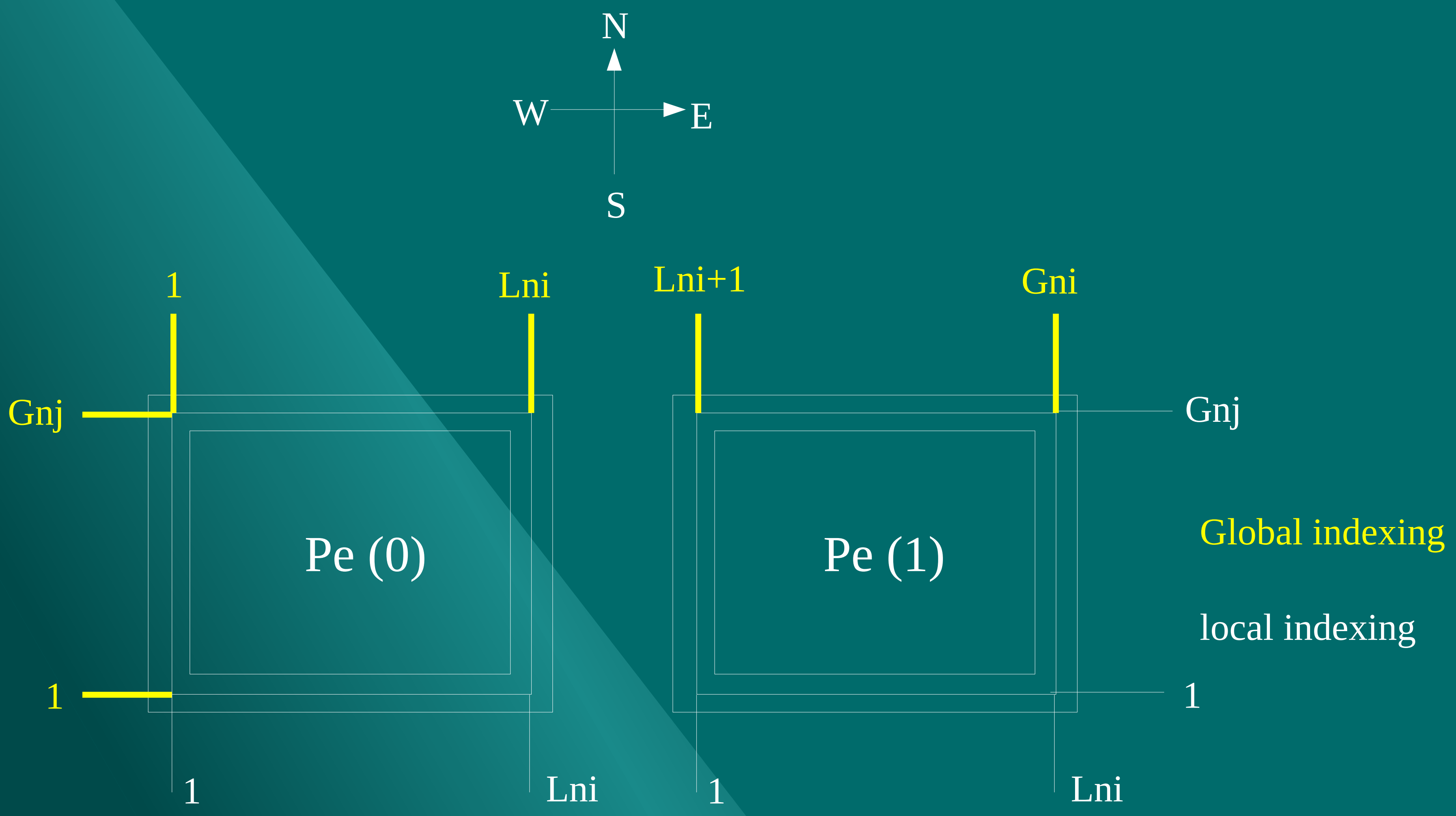


Domain decomposition

- Global (problem) topology
  - Each process (PE) only has a piece of the problem that represents a certain portion of the problem subscripting space
- Local topology
  - All processes use local subscripts to refer to their own data
  - Usually all processes use the same subscripting space for their own piece of the problem (storage / operation dimensions may vary as all pieces are not necessarily the same size)
  - Processes also need to know their position in the global problem (processor topology)

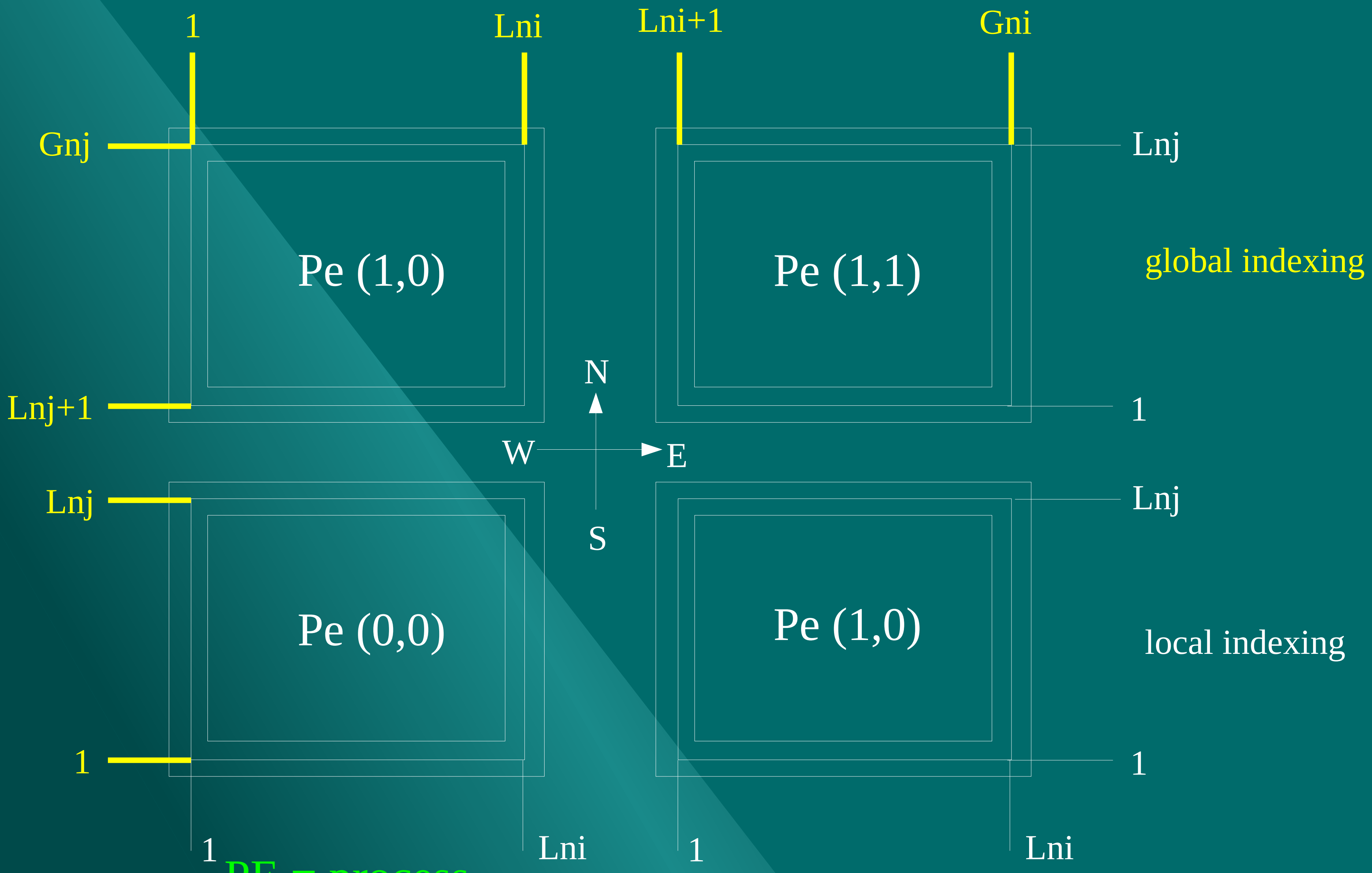
N.B. A process (PE) may use multiple threads.

1D domain decomposition



PE = process

2D domain decomposition

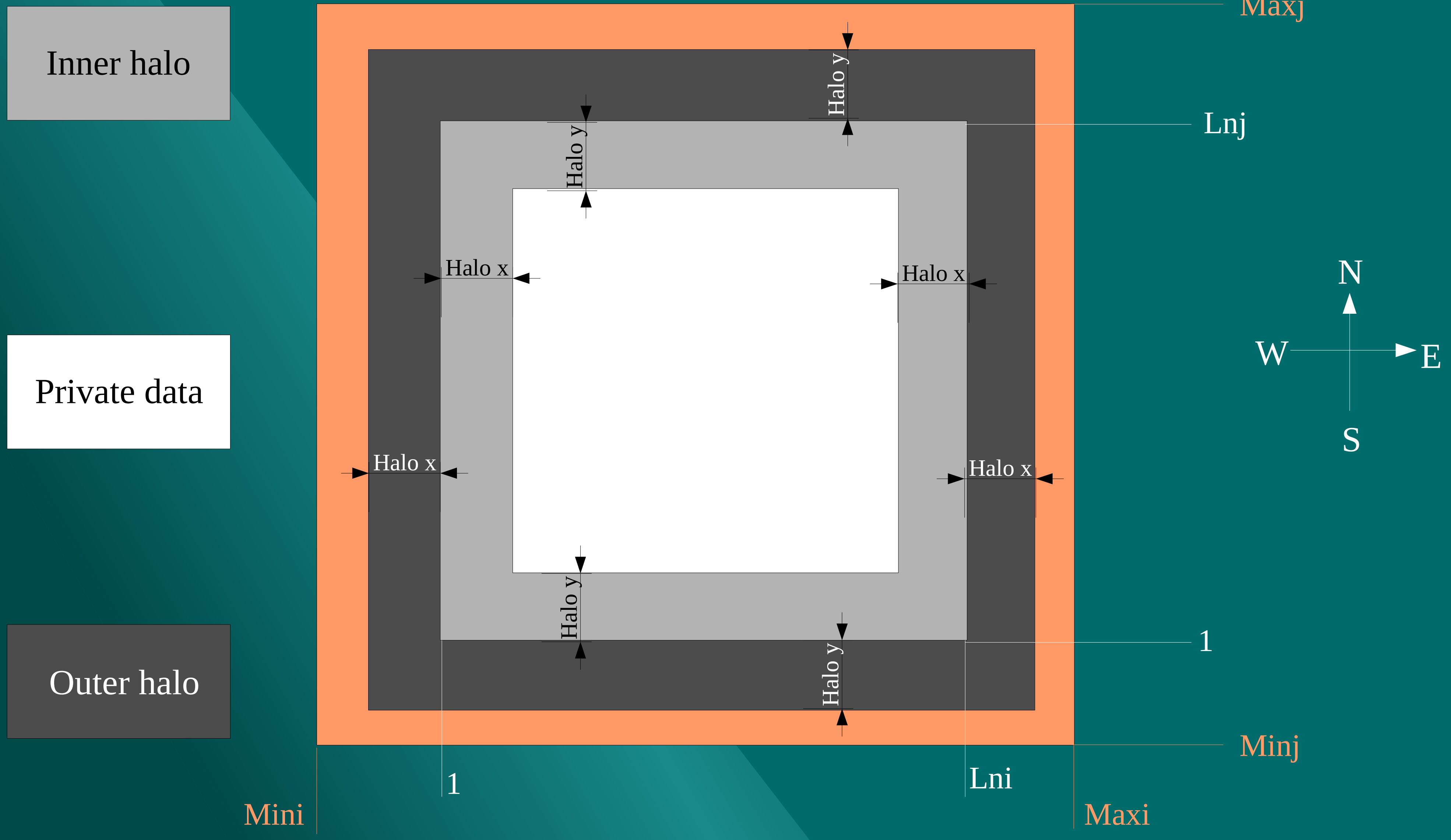


PE = process

High level operations

- Halo exchange
  - What is a halo ?
  - Why and when is it necessary to exchange a halo ?
- Data transpose
  - What is a data transpose ?
  - Why and when is it necessary to transpose data ?
- Reduction operations

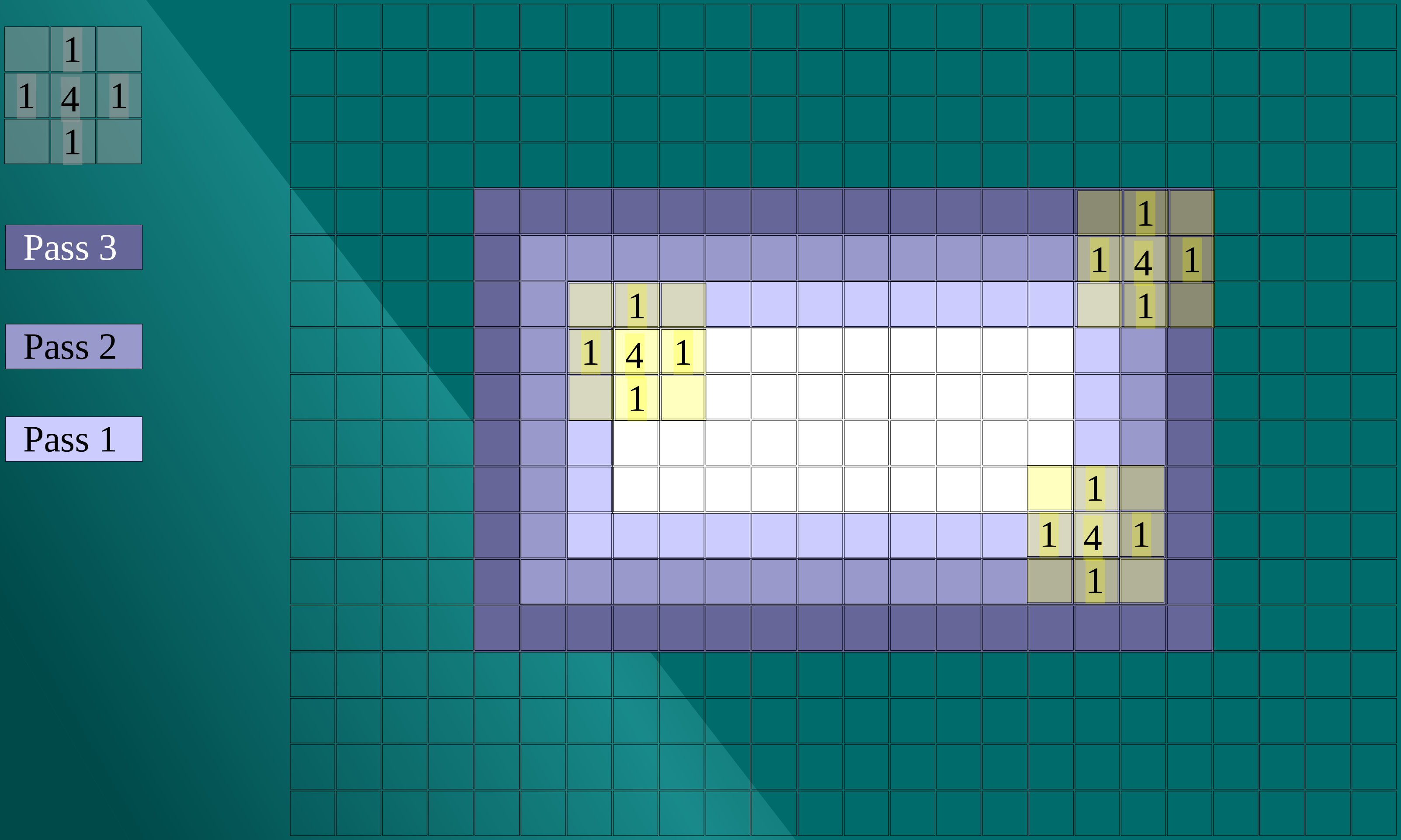
2D array layout with halos



Halo why and when ?

- Sometimes it is necessary to have access to neighboring data in order to perform local computa-tions
  - Differential operators
    dfdx(i) = (f(i+1) - f(i-1)) / (x(i+1)-x(i-1))
  - Filters
    new(i) = .25 * (old(i-1) + old(i+1) + 2*old(i))
  - In general any stencil type discrete operator
- Necessary halo width depends on the operator

2D smoothing

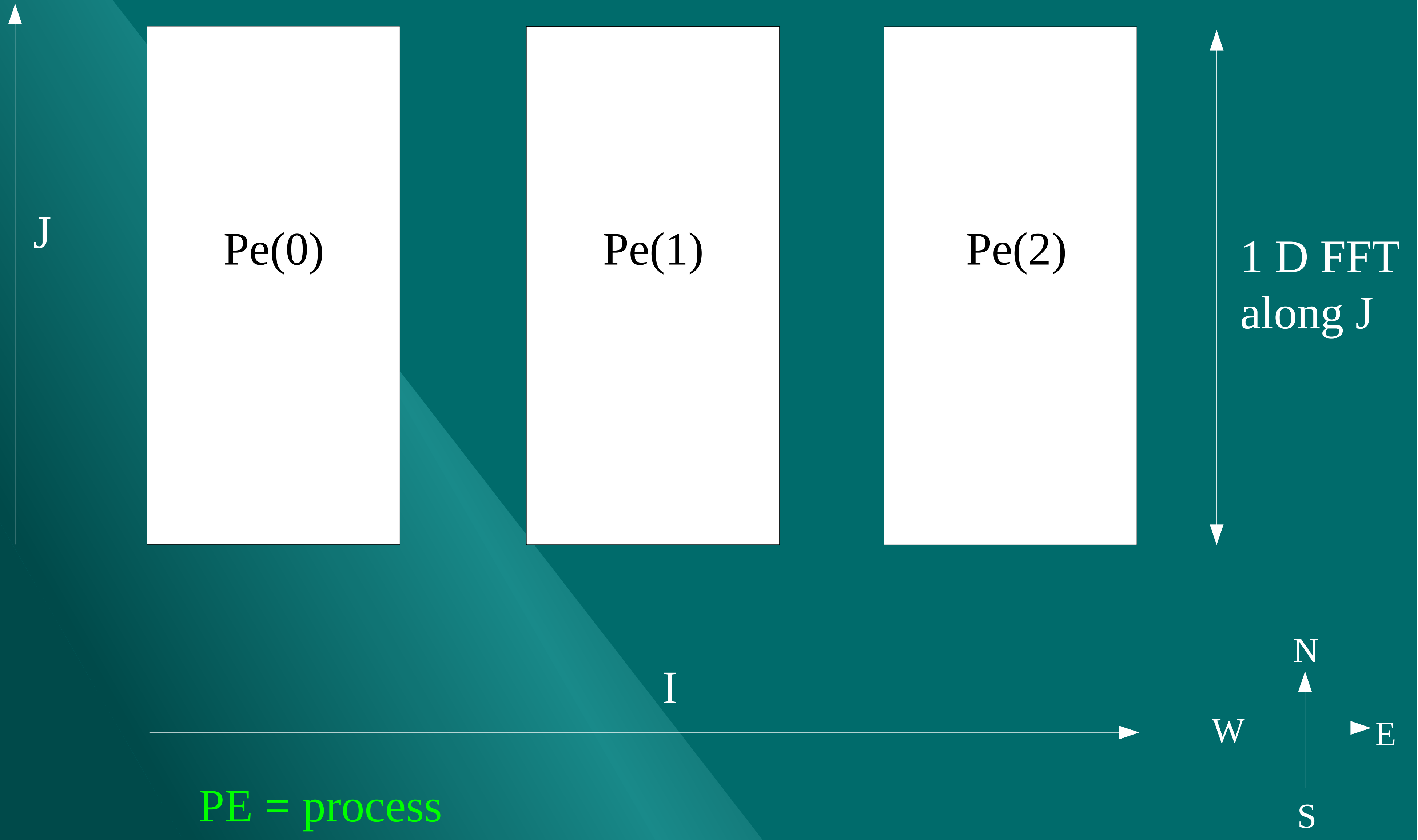


Halo width vs number of exchanges tradeoff

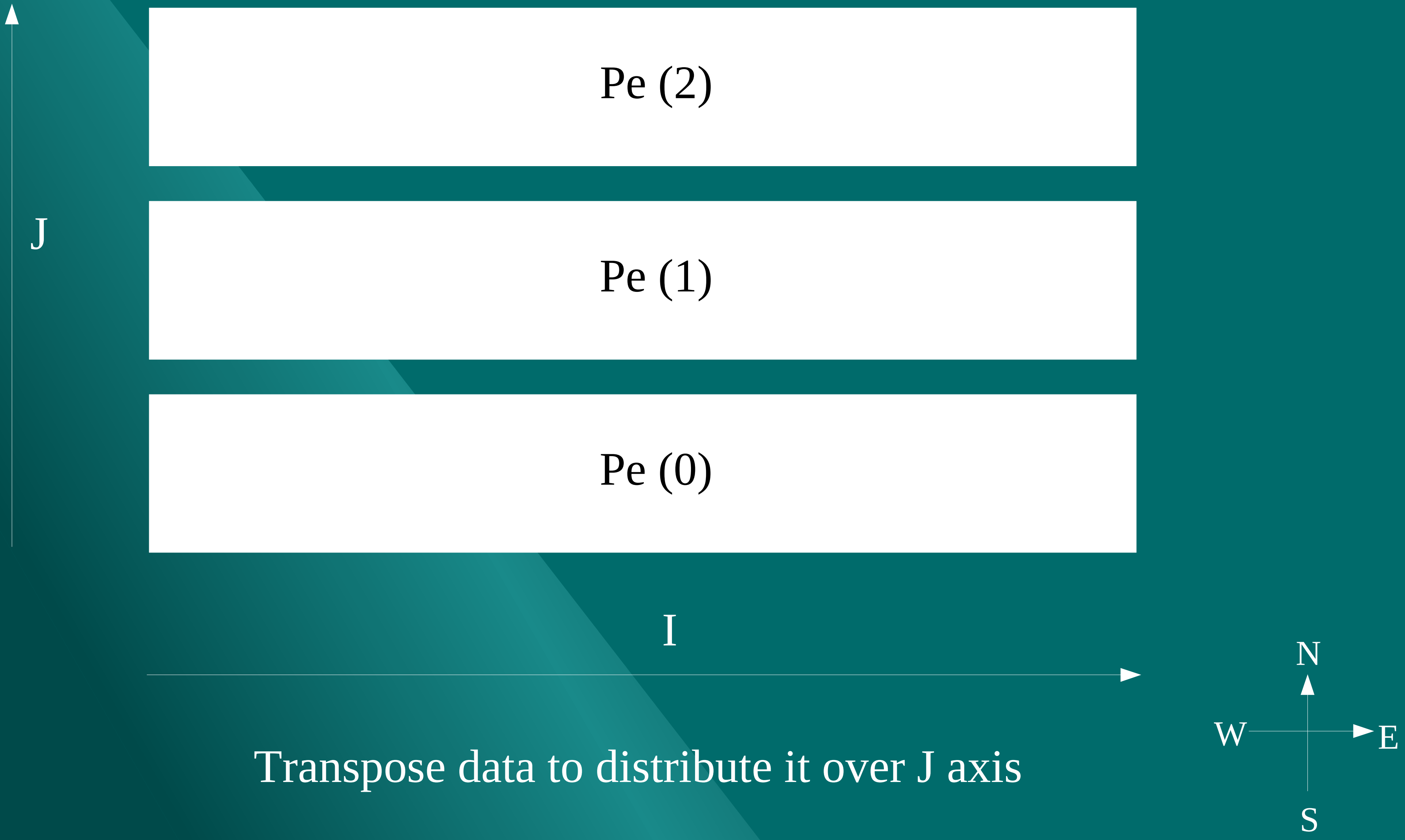
Data transpose

- A data transpose is a domain data decomposition change performed during the course of execution
- Suppose that we started with a 1D data decomposition (the I axis is distributed over processors)
- We now need to perform a 2D FFT over the data
  - Along J, no problem
  - Along I, OOPS !!
- Need to change data decomposition to bring I axis in processor for FFT along I

2 D FFT (step 1)

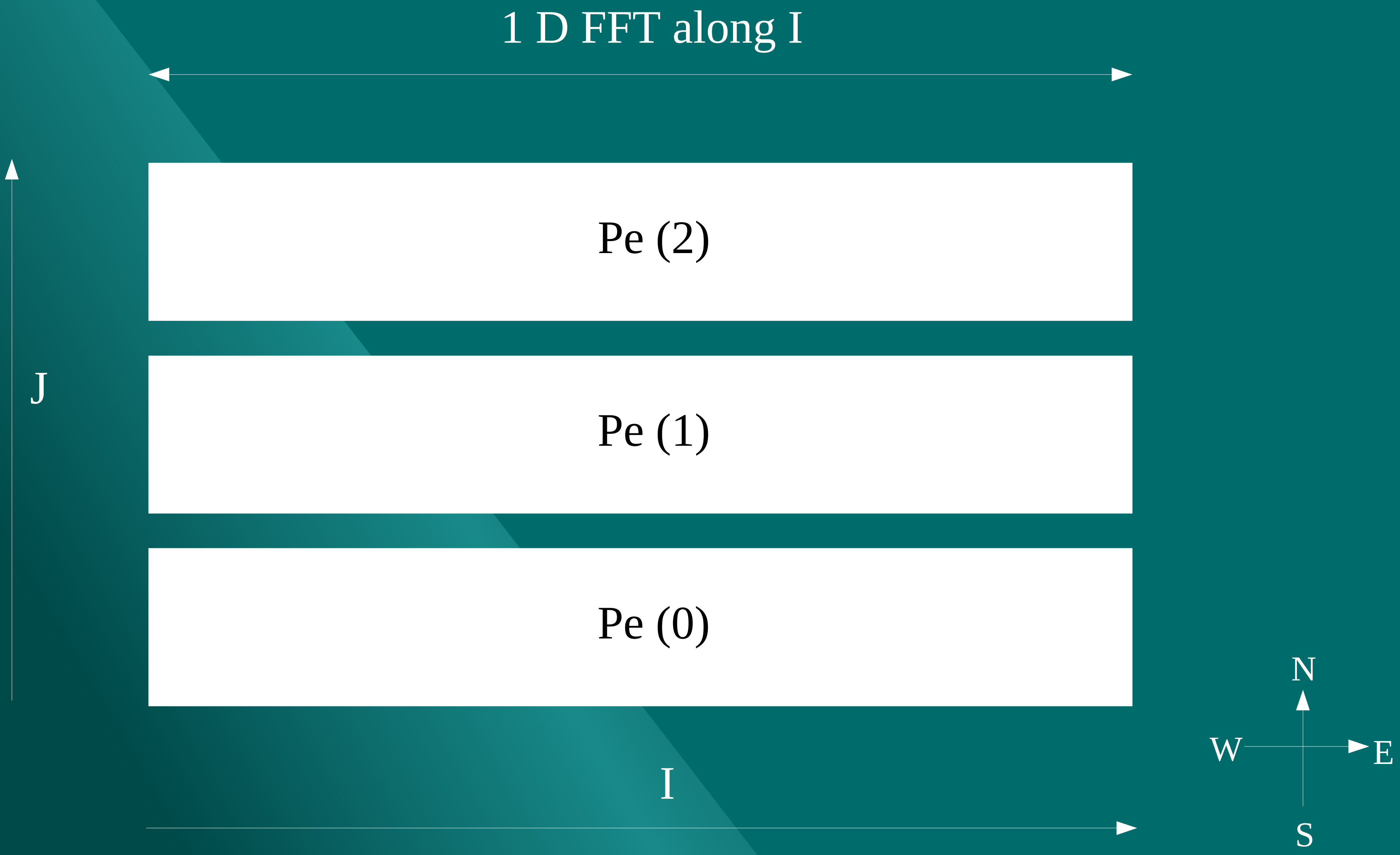


2 D FFT (step 2)

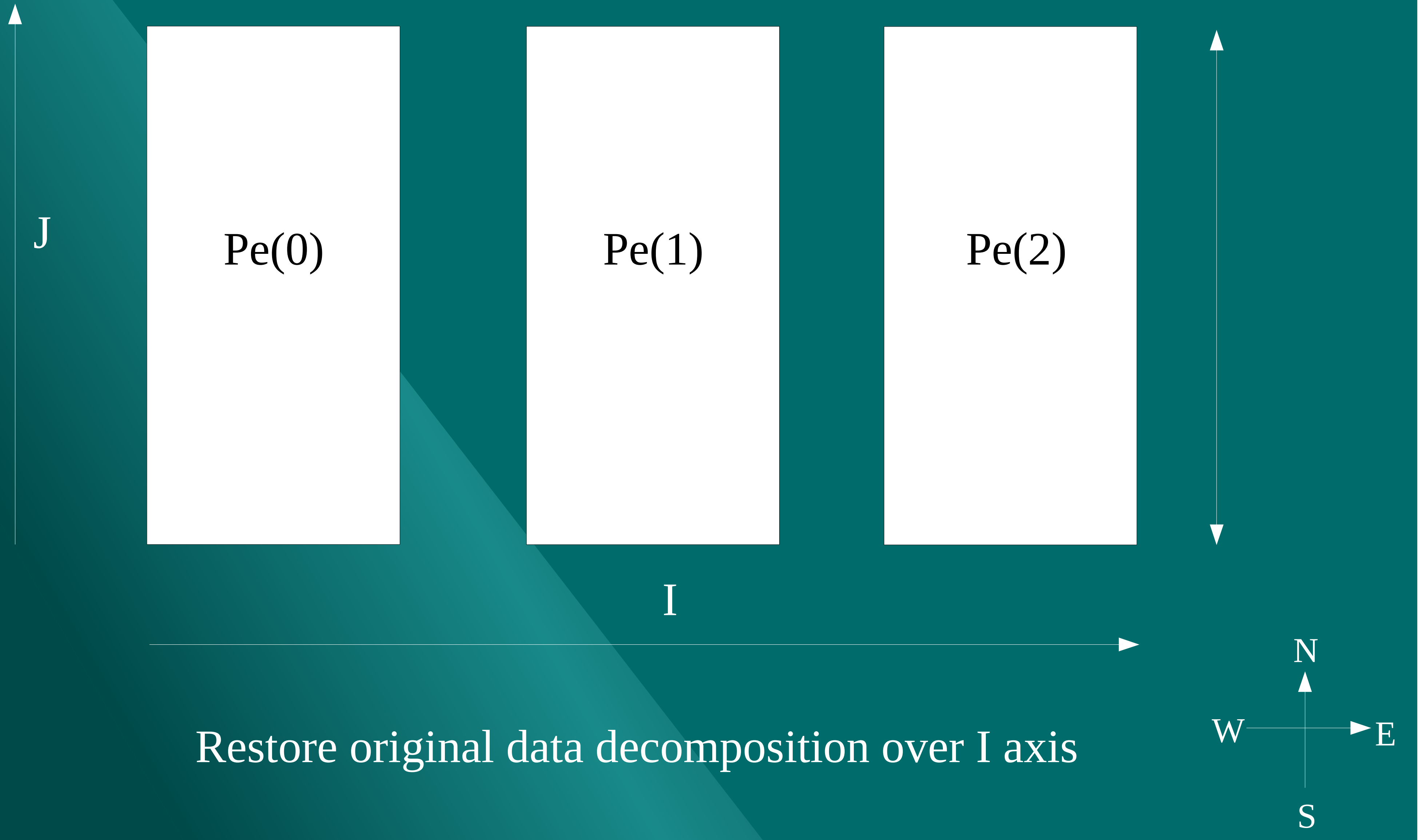


Transpose data to distribute it over J axis

2 D FFT (step 3)



2 D FFT (step 4)



Restore original data decomposition over I axis

Reduction operations

- Global reduction operations are sometimes needed
  - decomposition invariant
    - Min / max of entire problem
  - Non decomposition invariant
    - Global dot product
    - Sum / average / standard deviation of global field
- It is a 2 or 3 step process
  - Perform operation (computation) on local data
  - Use collective operator to perform a network reduction
  - Broadcast results to all processors if necessary

Reduction operations

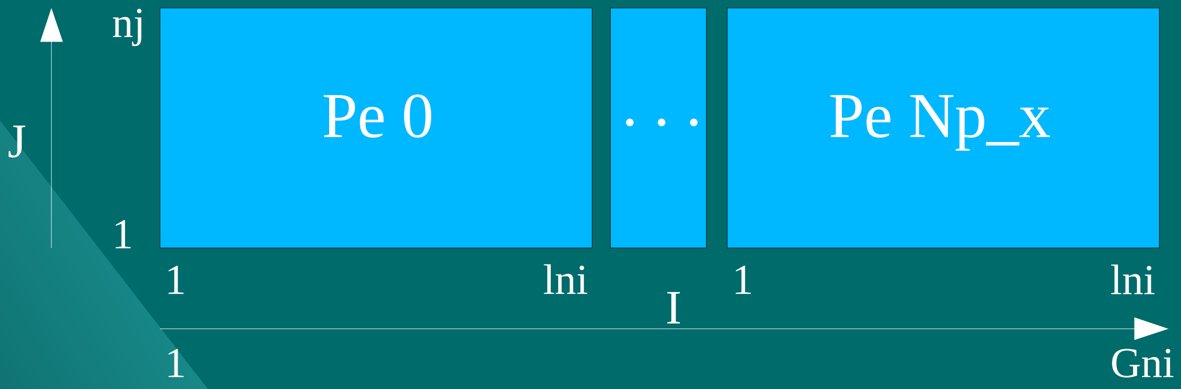
- What makes a reduction non decomposition invariant ?

  Ex: $(A + B) + C$ is not equal to $A + (B + C)$

- Can it be made decomposition invariant ?

  YES, but there is a butcher's bill

Reduction operation example
(sum along the I axis)



Method 1 (fast, non decomposition independent)

```
on each PE

localsum(:)=0
DO I=1,lni
  localsum(:)=localsum(:)+array(i,:)
ENDDO

MPI_all_reduce '+'  localsum
```

Method 2 (slow, decomposition independent)

```
on PE 0
   localsum(:)=0
on PE n
   get localsum from PE n-1

on  PE n
   DO I=1,lni
     localsum(:)=localsum(:)+array(i,:)
   ENDDO
   send localsum to PE n+1 (except for PE Np_x)

on PE Np_x
    MPI_broadcast  localsum
```

Method 2 keeps the summation order the same for any number of Pes at the expense of the parallelism

What is MPI ?

- A message passing library specification
  - Message passing model
  - Not a compiler specification
  - Not a specific product
- Bindings defined for FORTRAN, C, C++
- For parallel computers, clusters, heterogeneous networks
- Full featured (but can be used in simple fashion)

What is MPI ? (contd)

- Two part standard, MPI-1 and MPI-2
- Designed to permit the development of parallel software libraries
- Designed to provide access to advanced parallel hardware
  - End users
  - Library writers
  - Tool developers

Features of MPI

- General
  - Communicators combine context and group for security
  - Thread safety
- Point to point communications
  - Structured buffers and derived datatypes
  - Normal (blocking and non blocking) synchronous, ready (to allow for special fast protocols), buffered
- Collective communications
  - Built-in or user defined
  - Subgroups defined directly or by topology
  - Large number of data movement routines

Features of MPI (contd)

- Application oriented process topologies
  - Built-in support for groups and graphs
- Profiling
  - Hooks to allow users to intercept MPI calls and install their own tools
- environmental
  - Inquiry functions
  - Error control

Features not in MPI-1

- Non message-passing concepts not included:
  - Process management
  - Remote memory transfers (single sided communications)
  - Active messages
  - Threads
  - Virtual shared memory
- MPI does not address these issues but tries to remain compatible (e.g. Thread safety)
- Some of these features are in MPI-2

Is MPI large or small ?

- Mpi is large. MPI-1 has 128 functions, MPI-2 152
  - Extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many programs can be implemented with 6 basic functions
- MPI is just right
  - Functionality can be accessed when needed
  - No need to master all of MPI to use it

Where to use MPI ?

- You need a PORTABLE parallel program
- You are writing a parallel library (toolkit)
- You have irregular or dynamic data relationships that do not fit a data parallel model (e.g. HPF)
- You care about PERFORMANCE
  - Communications tend to degrade performance, lots of communica-tions mean lots of calls to MPI and make codes uglier. Beauty of the code can become a visual indicator of performance.

Where not to use MPI ?

- You can use HPF or a parallel FORTRAN 90 (one always hopes !!)
- OpenMP with a SMALL number of processors will be sufficient
- You can't care less about parallelism because problem is so small or speed is really not an issue
- You can directly use libraries (that may be written using MPI)
- Simple threading in a slightly concurrent environment is enough to save the day

MPI

- Writing MPI programs
- Compiling and linking
- Running MPI programs
- Examples can be found at :

  http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/

The FORTRAN 6 pack

- Include 'mpif.h'
- Call MPI_INIT(ierr)
- Call MPI_FINALIZE(ierr)
- Call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
- Call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
- Call MPI_SEND(buffer,count,datatype,destination,tag,comm,ierr)
- Call MPI_RECV(buffer,count,datatype,source,tag,comm,status,ierr)

Basic MPI program

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/basic.f

```
program hello
implicit none
include 'mpif.h'

integer noprocs, nid, error

call MPI_Init(error)
call MPI_Comm_size(MPI_COMM_WORLD, noprocs, error)
call MPI_Comm_rank(MPI_COMM_WORLD, nid, error)

write(6,*)'Hello from processor', nid, ' of',noprocs

call MPI_Finalize(error)

stop
end
```

Basic MPI program

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/basic.f

```
r.mpirun -npex 3 -pgm basic_Linux
```

```
 Hello from processor           0  of           3
 Hello from processor           1  of           3
FORTRAN STOP
 Hello from processor           2  of           3
FORTRAN STOP
FORTRAN STOP
```

Compiling, linking and running

To build executable :

```
r.compile -o my_program -src my_program.f -mpi
(  mpif90 -o my_program -src my_program.f  )
```

To run executable :

```
r.mpirun   -npex number_of_tasks    -pgm my_program
```

Compiling, linking and running

Makefile excerpt

```
basic_$(ARCH):
	r.compile -o basic_$(ARCH) -arch $(ARCH) -src basic.f -mpi
	rm basic.o

run_basic: basic_$(ARCH)
	r.mpirun -npex 3 -pgm basic_$(ARCH)

clean:
	rm -f basic_$(ARCH) basic.o
```

To build executable and run it on the current platform

```
make run_basic
```

Collective operations

- MPI_gather, MPI_allgather
- MPI_scatter, MPI_alltoall
- MPI_bcast

Pe 0	***Pe 1***	***Pe 2***	***Pe 3***	→	***Pe 0***	***Pe 1***	***Pe 2***	***Pe 3***
A	**B**	**C**	**D**	gather				**A,B,C,D**
A	**B**	**C**	**D**	allgather	**A,B,C,D**	**A,B,C,D**	**A,B,C,D**	**A,B,C,D**
		A,B,C,D		scatter	**A**	**B**	**C**	**D**
A,B,C,D	**E,F,G,H**	**I,J,K,L**	**M,N,O,P**	alltoall	**A,E,I,M**	**B,F,J,N**	**C,G,K,O**	**D,H,L,P**
	A			bcast	**A**	**A**	**A**	**A**
Send	***Send***	***Send***	***Send***	→	***Recv***	***Recv***	***Recv***	***Recv***

Reduction operations

- MPI operators:
  - mpi_sum
  - mpi_min, mpi_max
  - Other logicals operators
- mpi_reduce, mpi_allreduce

The RPN_COMM toolkit

(http://web-mrb.cmc.ec.gc.ca/mrb/si/eng/si/libraries/rpncomm/rpn_comm)

- NO INCLUDE FILE NEEDED (like mpif.h)
- Higher level of abstraction
- Initialization / termination of communications
- Topology determination
- Point to point operations
  - Halo exchange
  - (Direct message to NSWE neighbor)
- Collective operations
  - Transpose
  - Gather / distribute
  - Data reduction

The higher level operations

- Topology determination (local from global)
- Halo exchange
- Data transpose
- Data distribution
  - Collect / distribute
  - Broadcast
  - Reduction
- Neighbor to neighbor exchanges

DATA distribution (play by play)

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/allocate.f

```
Program halo
implicit none

include 'dimensions.h'

integer ierr
integer pelocal,petotal
external userinit
integer lni,lnj,mini,maxi,minj,maxj
integer lnimax,lnjmax,i0,j0,in,jn
integer lnpex,lnpey
integer ierr

real, dimension(:,:), allocatable :: z
```

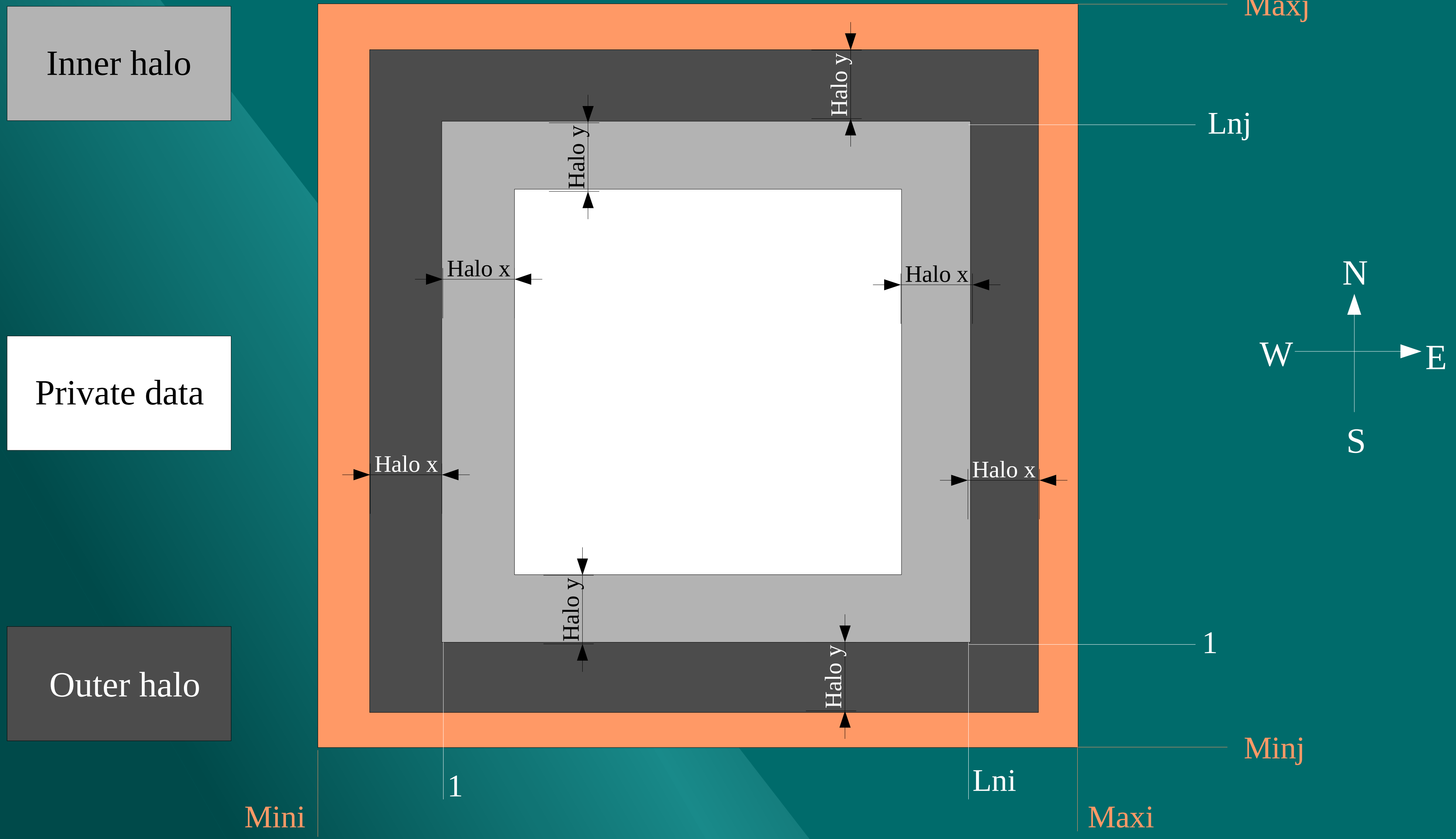
```
integer gni,gnj
integer npex,npey
integer halox,haloy
common /dimensions/gni,gnj,npex,npey,halox,haloy
namelist /dimensions/gni,gnj,npex,npey,halox,haloy
```

DATA distribution (contd)

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/allocate.f

```
      lnpex=0
      lnpey=0
*
*      INITIALIZE RPN_COMM
*
      call rpn_comm_init(userinit,pelocal,petotal,lnpex,lnpey)
*
*      BROADCAST run parameters
*
      call rpn_comm_bcast(gni,6,'MPI_INTEGER',0,'GRID',ierr)
*
*      determine TOPOLOGY
*
      ierr = rpn_comm_topo(gni,mini,maxi,lni,lnimax,halox,i0,.true.,.false.)
      in=i0+lni-1
      ierr = rpn_comm_topo(gnj,minj,maxj,lnj,lnjmax,haloy,j0,.false.,.false.)
      jn=j0+lnj-1
```

2D array layout with halos



DATA distribution (contd)
http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/allocate.f

```
*
*      ALLOCATE LOCAL portion of GLOBAL array
*
       allocate (z(mini:maxi,minj:maxj))

       print *, 'PE ',pelocal,' allocated z(',mini,':',maxi,
     $  ',',minj,':',maxj,')',' globalz(',i0,':',in,',',j0,':',jn,')',
     $  ' of (',gni,',',gnj,')'
*
*      TERMINATE gracefully
*
       call rpn_comm_finalize(ierr)
       stop
       end
```

DATA distribution (contd)

_http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/allocate.f_

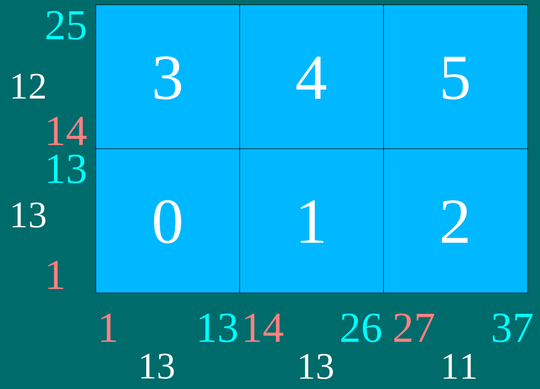
```
      integer gni,gnj
      integer npex,npey
      integer halox,haloy
      common /dimensions/gni,gnj,npex,npey,halox,haloy
      namelist /dimensions/gni,gnj,npex,npey,halox,haloy
```

```
       subroutine userinit(lnpex,lnpey)
      integer lnpex,lnpey
*
*      user initialization routine, called only on PE 0
*
      include 'dimensions.h'
      open(20,form='FORMATTED',file='indata_halo')
      read(20,nml=dimensions)
      write(6,nml=dimensions)
      close(20)
      lnpex=npex
      lnpey=npey
      return
      end
```

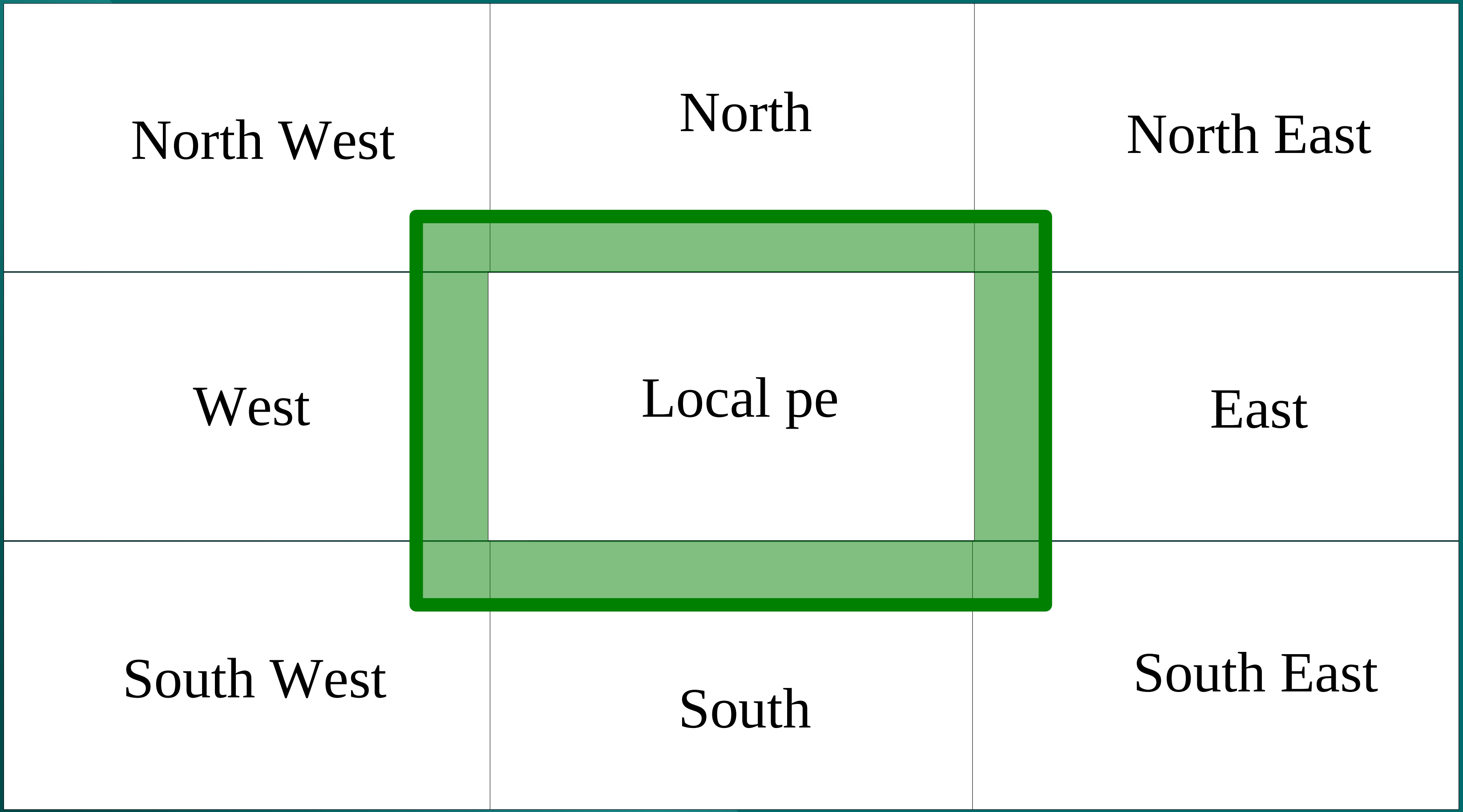
DATA distribution (contd)

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/HELLO/allocate.f

```
mpirun -np 6 allocate
 &DIMENSIONS  GNI = 37, GNJ = 25, NPEX = 3, NPEY = 2, HALOX = 2, HALOY = 3 /
 Requested topology =  3  by  2
 Domain set for  6  processes
 PE MATRIX :
   2   0   1   2   0
   5   3   4   5   3
   2   0   1   2   0
   5   3   4   5   3
 PE_xtab :
   0   1   2   0   1   2
 PE_ytab :
   0   0   0   1   1   1
 ordinals table
   0   1   2   3   4   5
 PE  0  allocated z( -1 : 15 , -2 : 16 ) globalz(  1 : 13 ,  1 : 13 ) of (1: 37 ,1: 25 )
 PE  1  allocated z( -1 : 15 , -2 : 16 ) globalz( 14 : 26 ,  1 : 13 ) of (1: 37 ,1: 25 )
 PE  2  allocated z( -1 : 15 , -2 : 16 ) globalz( 27 : 37 ,  1 : 13 ) of (1: 37 ,1: 25 )
 PE  3  allocated z( -1 : 15 , -2 : 16 ) globalz(  1 : 13 , 14 : 25 ) of (1: 37 ,1: 25 )
 PE  4  allocated z( -1 : 15 , -2 : 16 ) globalz( 14 : 26 , 14 : 25 ) of (1: 37 ,1: 25 )
 PE  5  allocated z( -1 : 15 , -2 : 16 ) globalz( 27 : 37 , 14 : 25 ) of (1: 37 ,1: 25 )
```



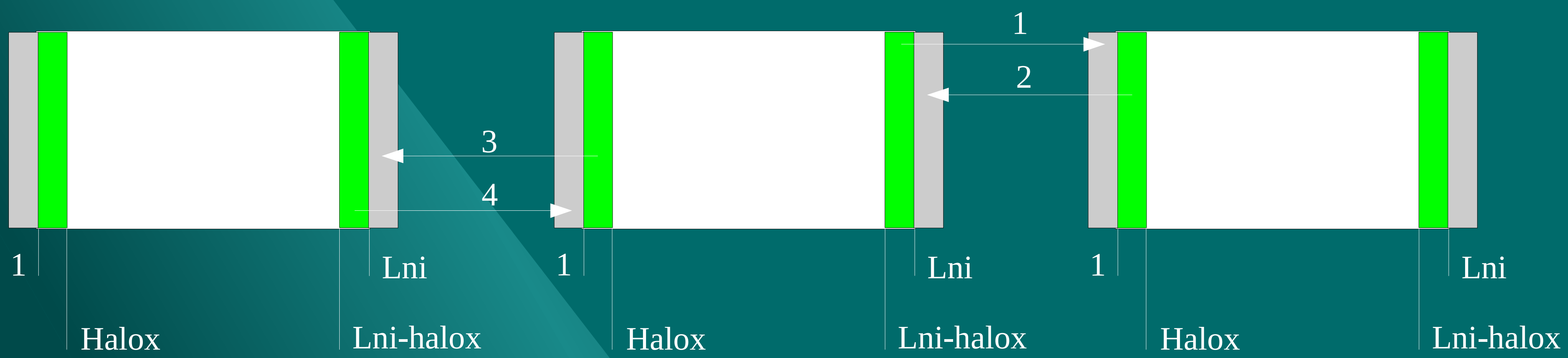
Halo exchange 051



How many neighbor PEs must local PE exchange data with to get data from the shaded area (outer halo, 8 neighbors)?

Halo exchange 101

Step 1, East-West exchange



🟩 Inner halo

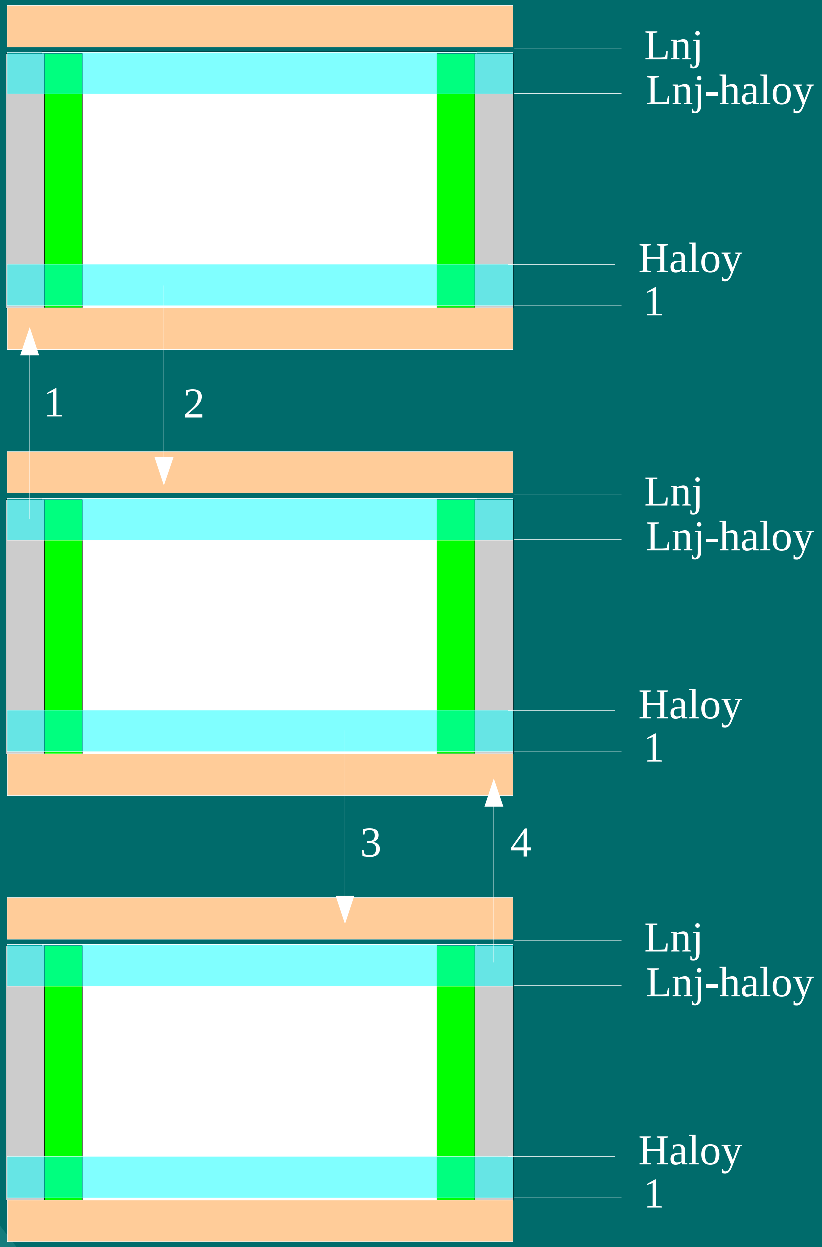
▨ Outer halo

1) Send East inner halo to East neighbor
2) Get East outer halo from East neighbor
3) Send West inner halo to West neighbor
4) Get West outer halo from West neighbor

Halo exchange 101

Step 2, North-South exchange

1) Send North inner halo to North neighbor
2) Get North outer halo from North neighbor
3) Send South inner halo to South neighbor
4) Get South outer halo from South neighbor



Inner halo (NS)	Inner halo (EW)
Outer halo (NS)	Outer halo (EW)

Other RPN_COMM tools

- Equivalent calls to most frequently used MPI routines
  - Send, Recv
  - Gather, Allgather, Reduce, Allreduce, Alltoall, Barrier
  - If you need one, ask for it!
- MPI_[something] => RPN_COMM_[something]

Example

- To send an array using MPI, you would use:

  call mpi_send(array, array_size, mpi_integer, dest, tag, MPI_COMM_world, ierr)

  dest is an absolute PE number

- With RPN_COMM, it becomes:

  call rpn_comm_send(array, array_size, ”mpi_integer”, “W”, tag, ”GRID”, ierr)

  May be used to target 'N', 'S', 'E', 'W' neighbors without having to know their PE number

Use of strings instead of MPI_ variables

- Datatypes:
  - mpi_integer => “mpi_integer”
  - mpi_real => “mpi_real”
  - Same for complex, double, etc...
- Operators
  - mpi_sum, max, min, etc...
- Basic communicators
  - “GRID” for all the calculation domain
  - “EW” for rows
  - “NS” for columns

Other RPN_COMM_tools

- Global collect, global distribute
  - RPN_COMM_coll
    - Retrieve data in a g_ni*g_nj array
  - RPN_COMM_dist:
    - Send data in a l_ni*l_nj array
- Global sum
  - Using rpn_comm_reduce with mpi_sum is not always a good idea
  - RPN_COMM_globalsum fixes the problem

It's your turn now!

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

- John Conway, game of life (1970)
  - http://www.math.com/students/wonders/life/life.html
  - It starts with an arbitrary initial pattern
  - The evolution follows some basic rules
  - Your universe can increase/decrease/die!

Rules of Life

- Your initial domain is filled with dead and living cells
- Each cell has eight neighbors
- A cell becomes alive if it has exactly three living neighbors
- A cell remains alive if it has two or three living neighbors
- Else, a cell dies or remains dead (loneliness or overcrowd-ing)

Remarks

- The results are computed from the state BEFORE the application of all rules.
- We use a 49 by 51 grid and suppose that everything outside the domain is dead and remains dead

Coding rules

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/LIFE/

- mpif.h not recommended (use RPN_COMM instead!)
- Routines game_of_life and show_results available from examples directory
  ~armnmfv/public_html/HPC_COURSE
  http://iweb.cmc.ec.gc.ca/~armnmfv/HPC_COURSE
  ~arnmbmk/PAR_WORKSHOP
- Keep track of the population count
- Stop experience if population <= 2 (why?)
- Don't cheat by looking at what your neighbor is doing, your idea may be good or better!

THE PROBLEM CODE

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/MPI/LIFE/

```
•  [TXT] REFERENCE_output     06-Oct-2006 20:36   2k  ( what you should get )
•  [TXT] RUN_MPI              06-Oct-2006 21:04   1k  ( RUN_MPI nx ny
                                                        to run with nx by ny topology)
•  [TXT] RUN_SINGLE           06-Oct-2006 20:55   1k  ( run the serial version )
•  [TXT] game_of_life.f90     06-Oct-2006 20:36   2k  ( the board processing and
                                                        display routines )
•  [TXT] life-serial.f90      06-Oct-2006 20:36   1k  ( the serial version )
•  [   ] lifempi2.f90         06-Oct-2006 20:36   4k  ( an answer )
•  [   ] lifempi2a.f90        06-Oct-2006 20:36   2k  ( a partial answer 1)
•  [   ] lifempi2b.f90        06-Oct-2006 20:36   1k  ( a partial answer 2)
•  [   ] lifempi2c.f90        06-Oct-2006 20:36   1k  ( a partial answer 3)
•  [   ] lifempi2d.f90        06-Oct-2006 20:36   1k  ( a partial answer 4)
•  [TXT] problem_data.cdk90   06-Oct-2006 20:36   1k  ( a needed module )
   ( some irrelevant entries omitted )
```

Single tile program:

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

(http://web-mrb.cmc.ec.gc.ca/mrb/si/eng/si/libraries/rpncomm/rpn_comm)

```
Program life
  implicit none
  integer, parameter :: gni=49, gnj=51, nstep=70, npts=5
  integer board(0:gni+1,0:gnj+1),i
  integer :: xarray(npts) = (/25,25,25,24,26/)
  integer :: yarray(npts) = (/23,24,25,23,24/)
  board=0
  do i=1,npts
     board(xarray(i),yarray(i))=1
  enddo
  call show_results(board,0,gni+1,0,gnj+1,1,gni,1,gnj)
  do i=1,nstep
     call game_of_life(board,0,gni+1,0,gnj+1,1,gni,1,gnj,1)
  enddo
  call show_results(board,0,gni+1,0,gnj+1,1,gni,1,gnj)
  print *,'THE END'
  stop
end Program life
```

Single tile program:

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

```
subroutine game_of_life(board,imin,imax,jmin,jmax,i0,ni,j0,nj,nstep)
!
! Core subroutine of Conway's Game of Life.
!
  integer imin,imax,jmin,jmax,ni,nj,nstep,i0,j0
  integer board(imin:imax,jmin:jmax)
  integer buf(i0:ni,j0:nj)
  integer i,j,n,sum,indice
  external cute_function
  integer cute_function
!
```

Single tile program:

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

```
!
  do n=1,nstep
     do j=j0,nj
        do i=i0,ni
           sum= board(i-1,j)   + board(i+1,j) + &
                board(i-1,j+1) + board(i+1,j+1) + board(i,j+1) + &
                board(i-1,j-1) + board(i+1,j-1) + board(i,j-1)
           if((board(i,j)==0).and.(sum==3)) then
              buf(i,j)=cute_function(1)
           else if ((board(i,j)==1).and.((sum==2).or.(sum==3))) then
              buf(i,j)=cute_function(1)
           else
              buf(i,j)=cute_function(0)
           endif
        enddo

     enddo
     board(i0:ni,J0:nj)=buf(i0:ni,j0:nj)
  enddo
```

Single tile program:

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

```
!
! put zero outside of the domain
!
  board(imin:i0-1,jmin:jmax)=0
  board(ni+1:imax,jmin:jmax)=0
  board(imin:imax,jmin:j0-1)=0
  board(imin:imax,nj+1:jmax)=0

end subroutine game_of_life
```

Single tile program:

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

```
integer function cute_function(input)
  integer input
  real temp
  integer itemp
  integer iter
  itemp=input
  do iter=1,200      ! really lose CPU time
    temp=itemp
    temp=asin(temp*.97)
    if(temp .lt. .4) temp = .05
    if(temp .gt. .5) temp=1.005
    itemp=nint(temp+.01)
  enddo
  if(input .ne. itemp) print *,' ERROR, cute_function is not identity'
  cute_function=itemp
return
end function cute_function
```

Single tile program:

http://iweb.cmc.ec.gc.ca/~armnmfv/COURS_HPC/SERIAL/LIFE/

```
subroutine show_results(board,imin,imax,jmin,jmax,i0,ni,j0,nj)
  integer imin,imax,jmin,jmax,ni,nj
  integer board(imin:imax,jmin:jmax)
  integer j
  character*1 out(2)
  out(1)='.'
  out(2)='X'
  write(*,*)
  do j=nj,j0,-1
     write(*,"(50A1)") out(board(i0:ni,j)+1)
!     write(*,"(50I1)") board(i0:ni,j)
  enddo
  write(*,*)

20 format('(80I1)')
end subroutine show_results
```

Suggestion for your code

- Before doing anything else, sketch a plan!
- Try to reduce the number of communications
- Hint: use halos where and when needed (watch out for global boundary conditions)

Debugging tips
(code does not always works on first try!!)

- Does it work for one PE (process)?
- Compile/execute your code often!
- Does the PE get the right message from the right sender?
- A great, fancy mpi debugger:

```
if(my_id==checked_pe) write(*,*) the_variables_I_want_to_check
call flush (6)
call rpn_comm_barrier
```

- Watch for zombie processes (ps -fu username)
  (N.B. You are all using the same username)

To make it better

- What could be done to avoid the halo exchange on each time step?
- Does your population count tracker use the most effi-cient collective operation?
- Bonus: on an infinite grid, do all patterns stabilize?

To make it better

- What could be done to avoid the halo exchange on each time step?
- Does your population count tracker use the most effi-cient collective operation?
- Bonus: on an infinite grid, do all patterns stabilize?
  - The answer is no, and a prize was awarded for the proof...

MPI: Conclusion
(expensive, but you get what you pay for)

- MPI is great if used correctly
- A software can't be ported easily to MPI if it is not designed for **PARALLEL** and **DISTRIBUTED** computation
- Monitor your jobs!
  - ps -fu username, all processes should advance at the same pace
  - If on a cluster, try to use similar (preferably identical) machines (the slowest process will set the overall speed)

THE END

Thank you for your attention

have (parallel) fun

may the (MPI) force be with you

Michel Valin, Luc Corbeil

Environnement Canada
Dorval, Québec