Parallel programming

V. Balaji and Rusty Benson
NOAA/GFDL and Princeton University

SSAM 2012
17 July 2012

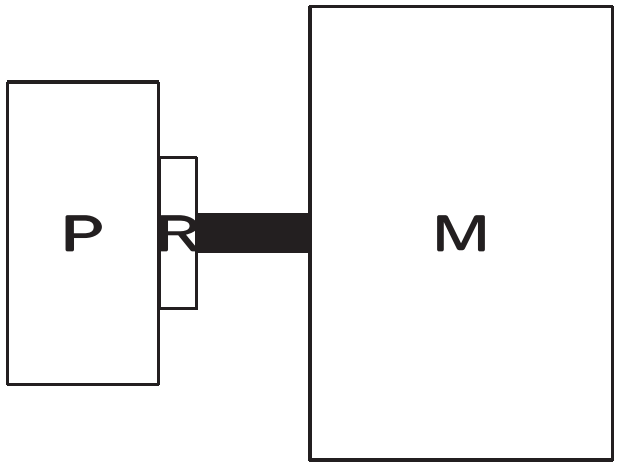
Overview

- Models of concurrency
- Overview of programming models: shared memory, distributed memory with remote memory access, message-passing
- Overview of the MPI programming interface
- Parallel programming considerations: communication and synchronization, domain decomposition
- Analyzing parallel algorithms: advection equation, Poisson equation
- Current research in parallel programming models

Sequential computing

The von Neumann model of computing conceptualizes the computer as consisting of a memory where instructions and data are stored, and a processing unit where the computation takes place. At each turn, we fetch an operator and its operands from memory, perform the computation, and write the results back to memory.

a = b + c



Computational limits

The speed of the computation is constrained by hardware limits:

- the rate at which instructions and operands can be loaded from memory, and results written back;

- and the speed of the processing units. The overall computation rate is limited by the slower of the two: memory.

Latency time to find a word.

Bandwidth number of words per unit time that can stream through the pipe.

Hardware trends

A processor clock period is currently $\sim$ 0.5-1 ns, “Moore’s Law” time constant is $4\times$/3 years.

RAM latency is $\sim$ 30 ns, Moore’s constant is $1.3\times$/3 years.

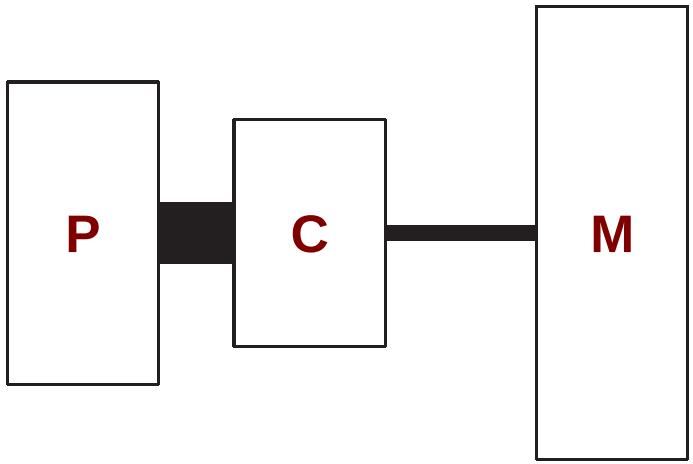
Maximum memory bandwidth is theoretically the same as the clock speed, but far less for commodity memory.

Furthermore, since memory and processors are built basically of the same “stuff”, there is no way to reverse this trend.

Caches

The memory bandwidth bottleneck may be alleviated by the use of caches.

Caches exploit **temporal locality** of memory access requests. Memory latency is also somewhat obscured by exploiting **spatial locality** as well: when a word is requested, adjacent words, constituting a **cache line**, are fetched as well.



Concurrency

Within the raw physical limitations on processor and memory, there are algorithmic and architectural ways to speed up computation. Most involve doing more than one thing at once.

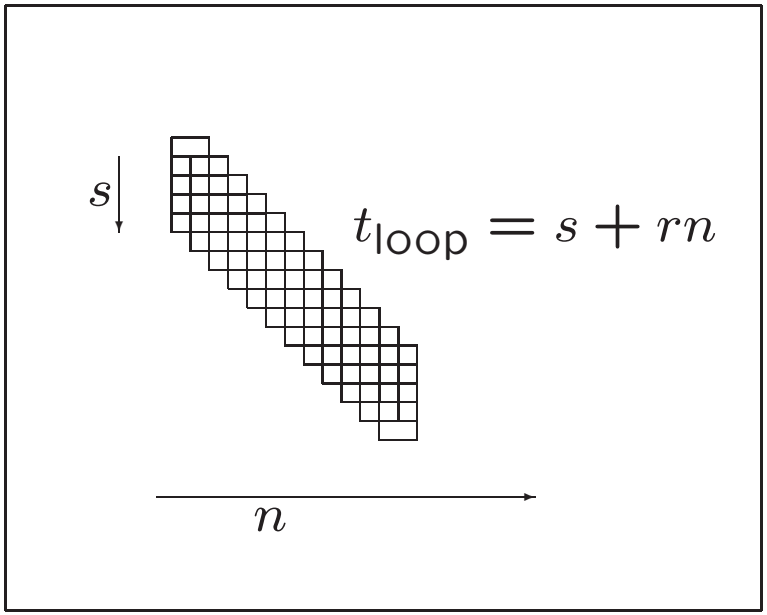
- Overlap separate computations and/or memory operations.
  - Pipelining.
  - Multiple functional units.
  - Overlap computation with memory operations.
  - Re-use already fetched information: **caching**.
  - Memory pipelining.
- Multiple computers sharing data.

The search for **concurrency** becomes a major element in the design of algorithms (and libraries, and compilers). Concurrency can be sought at different grain sizes.

Vector computing

Cray: if the same operation is ***independently*** performed on many different operands, schedule the operands to stream through the processing unit at a rate r = 1 per CP. Thus was born **vector processing**.

```
do i = 1,n
   a(i) = b(i) + c(i)
enddo
```



Vector computing: parallelism by pipelining

So long as the computations for each instance of the loop can be concurrently scheduled, the work within the loop can be made as complicated as one wishes.

The magic of vector computing is that for $s \gg rn$, $t_{\text{loop}} \approx s$ for any length n!

Of course in practice s depends on n if we consider the cost of fetching n operands from memory and loading the vector registers.

Vector machines tend to be expensive since they must use the fastest memory technology available to use the full potential of vector pipelining.

Task parallelism

Real codes in general cannot be recast as a single loop of n concurrent sequences of arithmetic operations. There is lots of other stuff to be done (memory management, I/O, etc.) Since sustained memory bandwidth requirements over an entire code are somewhat lower, we can let multiple processors share the bandwidth, and seek concurrency at a coarser grain size.

```
!$OMP DO PRIVATE(j)
do j = 1,n
   call ocean(j)
   call atmos(j)
end do
```

Since the language standards do not specify parallel constructs, they are inserted through compiler directives. Historically, this began with Cray microtasking directives. More recently, community standards for directives (**`!$OMP`**, see **`http://www.openmp.org`**) have emerged.

Instruction-level parallelism

This is also based on the pipelining idea, but instead of performing the same operation on a vector of operands, we perform ***different*** operations simultaneously on different data streams.

```
a = b + c
d = e * f
```

The onus is on the compiler to detect ILP. Moreover, algorithms may not lend themselves to functional parallelism.

Amdahl's Law

Even a well-parallelized code will have some serial work, such as initialization, I/O operations, etc. The time to execute a parallel code on P processors is given by

$$t_1 = t_s + t_{\|} \quad (1)$$

$$t_P = t_s + \frac{t_{\|}}{P} \quad (2)$$

$$\frac{t_1}{t_P} = \frac{1}{s + \frac{1-s}{P}} \quad (3)$$

where $s \equiv \frac{t_s}{t_1}$ is the serial fraction.

Speedup of a 1% serial code is at most 100.

Load-balancing

If the computational cost per instance of a parallel region unequal, the loop as a whole executes at the speed of the slowest instance (implicit synchronization at the end of a parallel region).

Work must be partitioned in a way that keeps the load on each parallel leg roughly equal.

If there is sufficient granularity (several instances of a parallel loop per processor), this can be automatically accomplished by implementing a global task queue.

```
!$OMP DO PRIVATE(j)
do j = 1,n
   call ocean(j)
end do
```

Scalability

Scalability: the number of processors you can usefully add to a parallel system. It is also used to describe something like the degree of coarse-grained concurrency in a code or an algorithm, but this use is somewhat suspect, as this is almost always a function of problem size.

Weak scalability Code is scalable by increasing problem size.

Strong scalability Scalable at any problem size.

A general communication and synchronization model for parallel systems

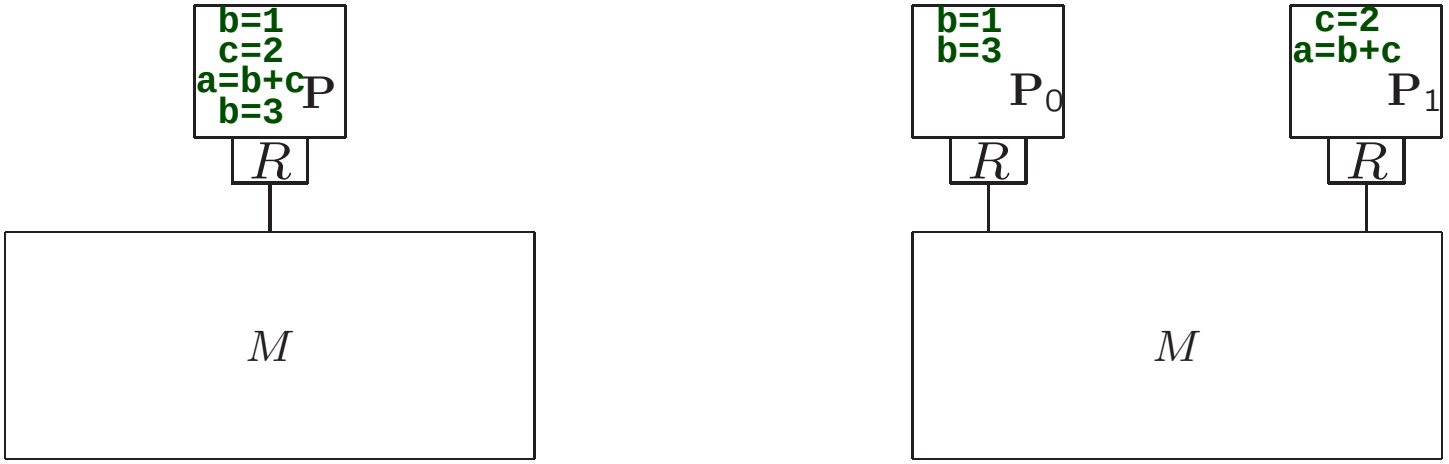
We use the simplest possible computation to compare shared and distributed memory models. Consider the following example:

```
real ::  a, b=0, c=0
b = 1
c = 2
a = b + c
b = 3
```

(4)

at the end of which both **a** and **b** must have the value 3.

Sequential and parallel processing

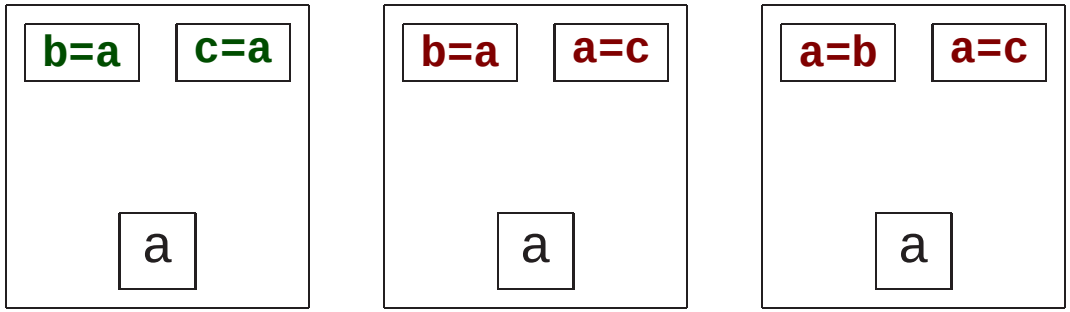


Let us now suppose that the computations of **b** and **c** are expensive, and have no mutual dependencies.

Then we can perform the operations ***concurrently***:

- Two PEs able to access the same memory can compute **b** and **c** independently, as shown on the right.
- Memory traffic is increased: to transfer **b** via memory, and to control the contents of cache.
- Signals needed when **b=1** is complete, and when **a=b+c** is complete: otherwise we have a ***race condition.***

Race conditions

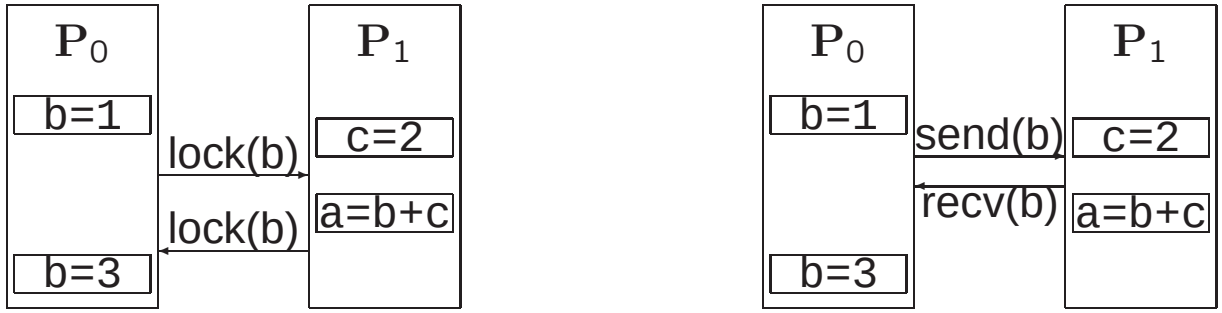


Race conditions occur when one of two concurrent execution streams attempts to write to a memory location when another one is accessing it with either a read or a write: it is not an error for two PEs to read the same memory location simultaneously. The second and third case result in a race condition and unpredictable results. The third case may be OK for certain reduction or search operations, defined within a **critical region.**

The central issue in parallel processing is the ***avoidance of such a race condition with the least amount of time spent waiting for a signal***: when two concurrent execution streams have a mutual dependency (the value of **a**), how does one stream know when a value it is using is in fact the one it needs? Several approaches have been taken.

Shared memory and message passing

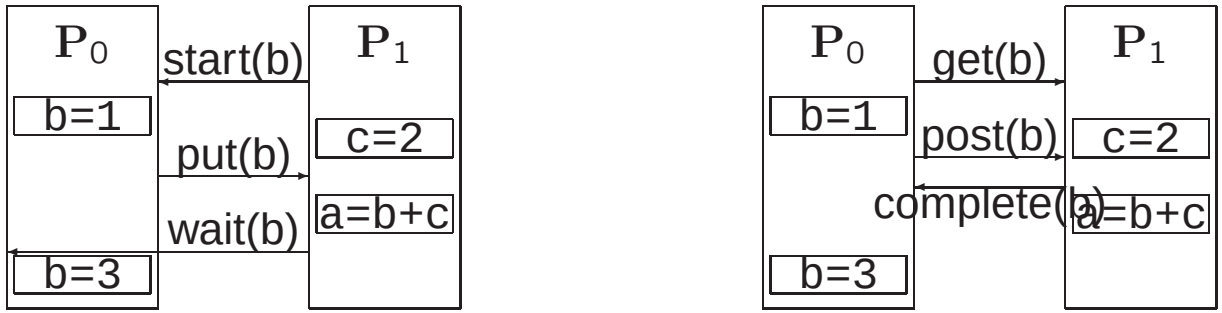
The computations **b=1** and **c=2** are concurrent, and their order in time cannot be predicted.



- In shared-memory processing, ***mutex locks*** are used to ensure that **b=1** is complete before P_1 computes **a=b+c**, and that this step is complete before P_0 further updates **b**.

- In message-passing, each PE retains an independent copy of **b**, which is exchanged in paired send/receive calls. After the transmission, P_0 is free to update **b**.

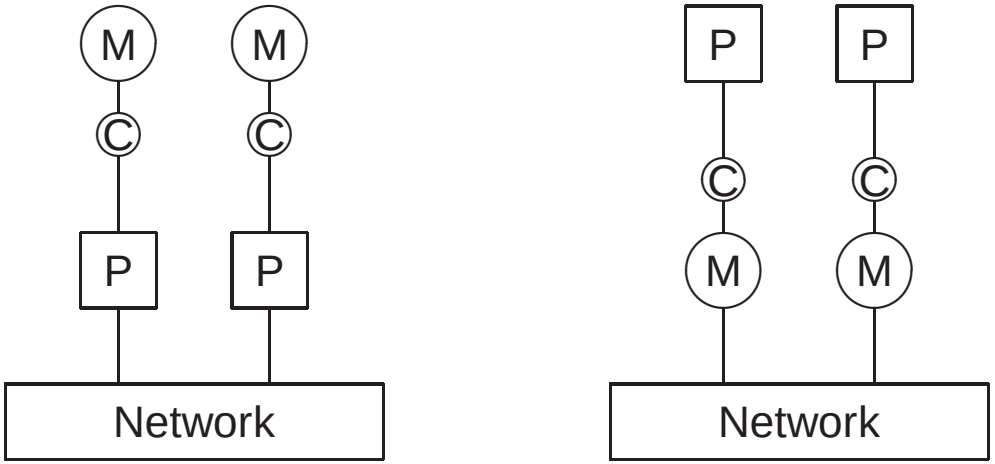
Remote memory access (RMA)



The name ***one-sided message passing*** is often applied to RMA but this is a misleading term. Instead of paired send/receive calls, we now have transmission events on one side (**`put`**, **`get`**) paired with ***exposure*** events (**`start`**,**`wait`**) and (**`post`**,**`complete`**), respectively, in MPI-2 terminology, on the other side. It is thus still “two-sided”. A variable exposed for a remote **`get`** may not be written to by the PE that owns it; and a variable exposed for a remote **`put`** may not be read.

Note that P_1 begins its exposure to receive **`b`** even before executing **`c=2`**. This is a key optimization in parallel processing, ***overlapping computation with communication***.

Non-blocking communication



- On ***tightly-coupled systems***, independent network controllers can control data flow between disjoint memories, without involving the processors on which computation takes place. True non-blocking communication is possible on such systems.

- Note that caches induce complications.

- On ***loosely-coupled systems***, this is implemented as the semantically equivalent ***deferred communication***, where a communication event is registered and queued, but only executed when the matching block is issued.

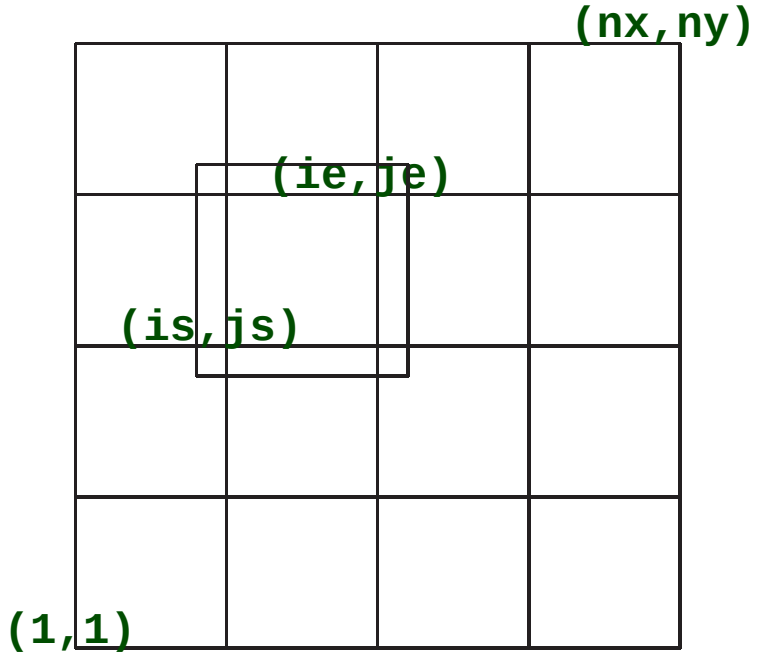
Memory models

Shared memory signal parallel and critical regions, private and shared variables. Canonical architecture: UMA, limited scalability.

Distributed memory domain decomposition, local caches of remote data ("halos"), copy data to/from remote memory ("message passing"). Canonical architecture: NUMA, scalable at cost of code complexity.

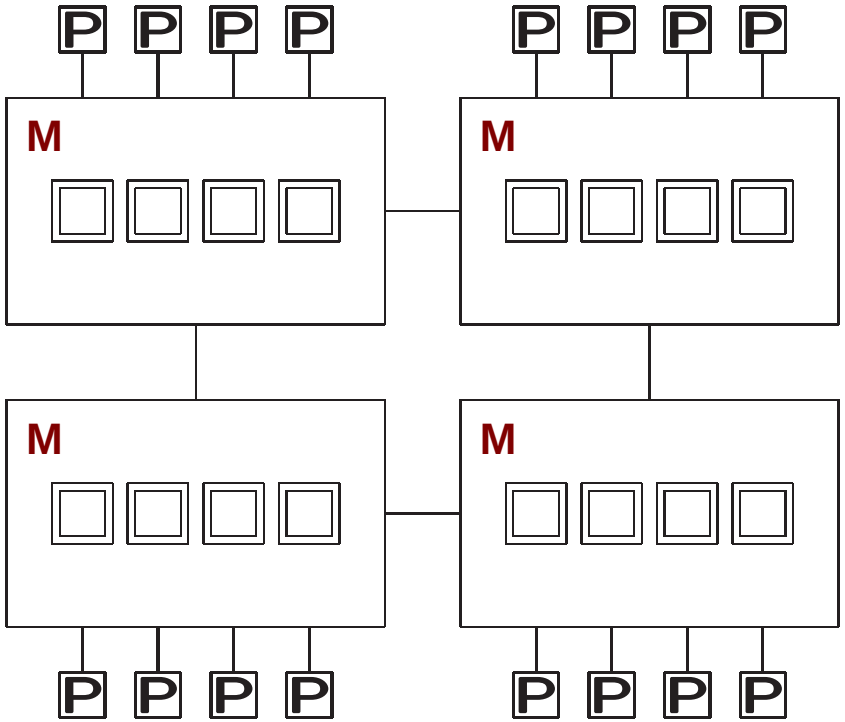
Distributed shared memory or ccNUMA message-passing, shared memory or remote memory access (RMA) semantics. Processor-to-memory distance varies across address space, must be taken into account in coding for performance. Canonical architecture: cluster of SMPs. Scalable at large cost in code complexity.

A 2D example



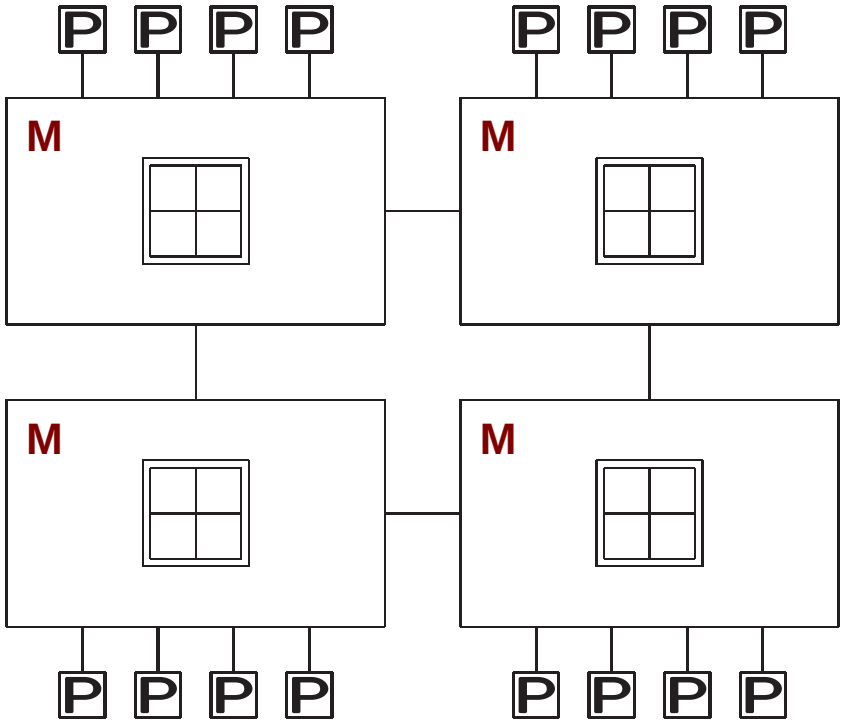
Consider a platform consisting of 16 PEs consisting of 4 **mNodes** of 4 PEs each. We also assume that the the entire 16-PE platform is a DSM or ccNUMA **aNode**. We can then illustrate 3 ways to implement a `DistributedArray`. One process is scheduled on each PE.

Distributed memory



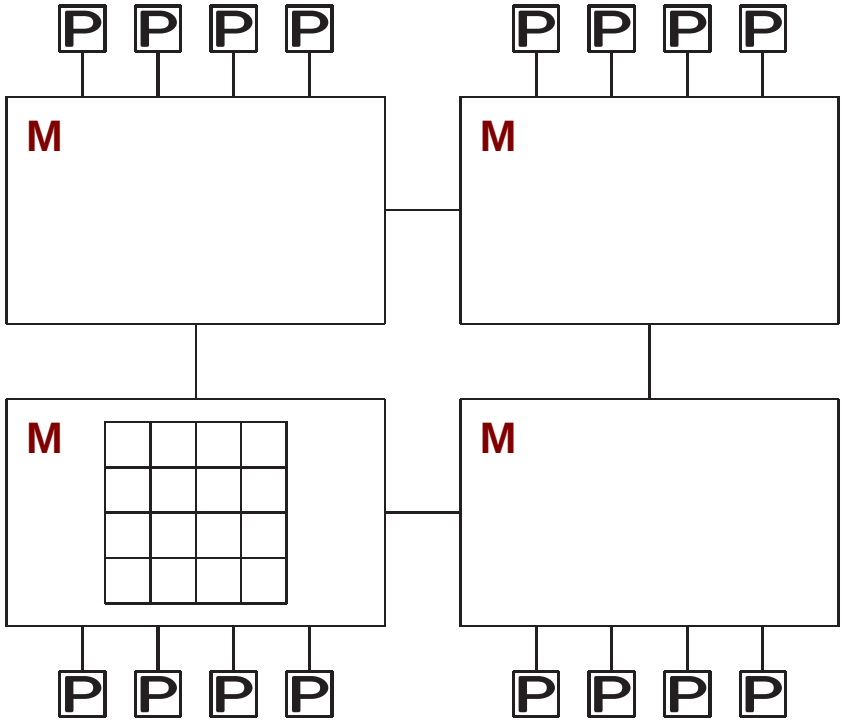
- each domain allocated as a separate array with halo, even within the same mNode.
- Performance issues: the message-passing call stack underlying MPI or another library may actually serialize when applied within an mNode.

Hybrid memory model



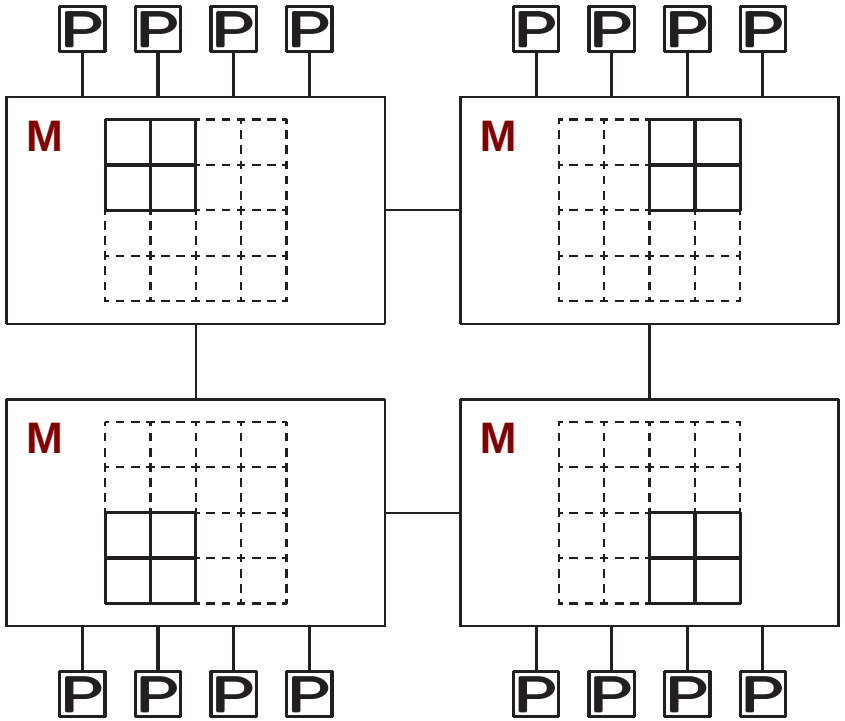
- shared across an mNode, distributed among mNodes.
- fewer and larger messages than distributed memory (communication/computation scales as surface/volume), may be less latency-bound.

Pure shared memory



Array is local to one mNode: other mNodes requires remote loads and stores. OK on platforms that are well-balanced in bandwidth and latency for local and remote accesses. **ccNUMA** ensures cache coherence across the aNode.

Intelligent memory allocation on DSM



Better memory locality: allocate each block of 4 domains on a separate page, and assign pages to different mNodes, based on processor-memory affinity.

Remote memory access with barriers

- Every PE has a unique ID, **me**.
- All PEs must arrive at a barrier before any PE can move on. The order in which they arrive is not predictable.

```
program test
integer :: me, right
me = my_pe()
call BARRIER()
call GET( right, me, 1, mod(me+1,npes) )
call BARRIER()
print *, 'PE', me, ' says hi to its neighbour on the right,', right
end
```

```
 PE 2  says hi to its neighbour on the right, 3
 PE 0  says hi to its neighbour on the right, 1
 PE 3  says hi to its neighbour on the right, 0
 PE 1  says hi to its neighbour on the right, 2
```

RMA synchronization

`get()` synchronization:

```
b = ...
call BARRIER()
call GET( a, b, 1, remote_PE )
(useful work not dependent on a)
call BARRIER()
... = a
```

`put()` synchronization:

```
b = a
call PUT( a, b, 1, remote_PE )
(useful work not dependent on a)
call BARRIER()
a = ...
```

- **`put()`** and **`get()`** are ***non-blocking***: return control to the sender after initiating communication.
- **`barrier()`** is a ***blocking*** operation.

Communication and synchronization

RMA is conceived with a tightly-coupled MPP in mind, where memory is close to the network. This permits the PE wishing to **`get()`** or **`put()`** data to a remote PE to proceed without interrupting the remote PE (if the hardware permits).

This requires a synchronization operation to make sure the transmitted data is available for the operation. Synchronization is effected with a **`barrier()`** call. On loosely-coupled systems barriers can be very expensive.

Distributed shared memory

More recently, with the advent of fast ***hardware cache-coherency*** techniques, the single-address-space programming model has been revived within the **cc-NUMA** architectural model. Here memory is ***physically*** distributed, but ***logically*** shared. Since it still involves remote memory access (though perhaps hidden from the user), distributed memory is still a correct lens through which to view its performance.

```
b = ...
(useful work not dependent on a)
call BARRIER()
a = b
call BARRIER()
... = a
```

MPI: a communication model for loosely-coupled systems

Loosely-coupled systems built out of commodity components now dominate the scene.

For a loosely-coupled or heterogeneous system, direct operations to a remote memory cannot be permitted.

The communication model is a **rendezvous**.

```
call GET( b, a, 1, from_pe ) !on to_pe
```

becomes

```
call MPI_SEND( a, ..., to_pe,   ... ) !on from_pe
call MPI_RECV( b, ..., from_pe, ... ) !on to_pe
```

There is now another level of latency – **software latency** – in negotiating the communication.

Evolution of MPI

MPI was originally developed in an era when “the network is the computer” was the prevailing ideology.

Many algorithms and problems are not loosely coupled, however. And at the high-end, on tightly-coupled hardware, the software overhead. of MPI became apparent.

Later extensions (MPI-2) offered an implementation of RMA that could also run on loosely-coupled systems, but could be implemented efficiently on the right architecture, with overheads comparable to native libraries, such as SHMEM.

MPI-2 also provided a layer for expressing I/O from distributed data in succinct ways. Implementations suffer from the same problems as parallel I/O in general faces.

Review of parallel programming models

- What is a ***race condition***?
- What is a ***tightly-coupled parallel system***?
- What is a ***barrier***?
- What is ***cache coherency***?

The MPI API: instantiation

Basic instantiation includes starting and stopping parallel execution, and to have each process uniquely identify itself and others.

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

int main(int argc, char **argv)
{
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("I am PE %d of %d\n", rank, size);

  MPI_Finalize();
}
```

```
mpirun -np 2 a.out
```

Each MPI call has a context called a ***communicator*** with a certain size, and each process has a unique ***rank*** within it. To address a message to another PE, both are needed.

MPI: blocking send and receive

Blocked messages are the simplest mode of communication.

```
void *buf;
int count, dest, tag;
MPI_Datatype type;
MPI_Comm comm;
MPI_Status status;

MPI_Send( buf, count, type, pe, tag, comm);
MPI_Recv( buf, count, type, pe, tag, comm, status);
```

`count` words of type **`type`** are sent or received from **`pe`** within the context **`comm`**.
`tag` is a user-supplied identifier for the message.

`type` can be a basic type (**`MPI_INTEGER, MPI_FLOAT`**, ...) or a more complex derived datatype for a complex message.

Problems with blocked communication: deadlock

- On PE 0:

```
MPI_Send( buf, count, type, 1, tag, comm);
MPI_Recv( buf, count, type, 1, tag, comm, &status);
```

- On PE 1:

```
MPI_Send( buf, count, type, 0, tag, comm);
MPI_Recv( buf, count, type, 0, tag, comm, &status);
```

The **`send()`** on PE 0 cannot complete until PE 1 calls **`recv()`**; and vice versa. Reversing the order of **`send/recv`** on one of the PEs will work.

Under the covers, MPI is using internal buffers to cache messages. A blocked comm pattern may work for some values of **`count`**, and then fail as **`count`** is increased.

MPI: non-blocking send and receive

A solution is to make at least one of **send/recv** non-blocking. A non-blocking call returns control to the caller after initiating communication. The status of the message buffer is undefined until a corresponding **wait()** call is posted to check the status of the message.

- On PE 0:

```
MPI_Request request;
MPI_Isend( buf, count, type, 1, tag, comm, &request);
... // other work that does not modify or free buf
MPI_Wait( &request, &status );
if( status == MPI_OK )
   buf = ...
```

- On PE 1:

```
MPI_Irecv( buf, count, type, 0, tag, comm, &request);
... // other work that does not require the contents of buf
MPI_Wait( &request, &status );
if( status == MPI_OK )
   ... = buf ...
```

MPI_Wait() is a blocking call. **MPI_Test()** can be used as an alternative to check if the pending communication is complete, without blocking.

MPI API summary

The basic calls within MPI have been described:

- instantiation: **`init`**, **`finalize`**, **`comm_rank`**, **`comm_size`**, . . .
- communication: **`send`**, **`recv`**, **`isend`**, **`irecv`**, . . .
- RMA: **`start/put/complete`**, **`post/get/wait`** . . .

Other aspects of the API include:

- Creation of communicators and intracommunicators (MPI-2)
- Broadcast, gather, scatter
- Reduction operations (reduce, allreduce);
- etc.

The API is vast!

A programming model for MPPs

The model we will be looking at here consists of:

- ***Distributed***, as opposed to shared, memory organization.
- ***Local***, as opposed to global, address space.
- ***Non-uniform***, as opposed to uniform, memory access.
- ***Domain decomposition***, as opposed to functional decomposition.

Parallel programming model

A ***task*** is a sequential (or vector) program running on one processor using local memory.

A parallel computation consists of two more tasks executing concurrently.

Execution does not depend on particular assignment of tasks to processors. (More than one task may belong to a processor.)

Tasks requiring data from each other need to synchronize and exchange data as required. (We do not consider ***embarrassingly parallel*** problems here, where there is no need for synchronization and data exchange.)

Partitioning

Issues to consider in partitioning the problem into tasks:

- Data layout in memory.
- Cost of communication.
- Synchronization overhead.
- Load balance.

Communication model

A ***message*** consists of a block of data ***contiguously laid out in memory***.

Communication consists of an non-blocking **`send()`** and a blocking **`recv()`** of a message. In loosely-coupled systems, PEs need to negotiate the communication, thus both a **`send()`** and a **`recv()`** are required. In tightly-coupled systems we can have a pure **`send()`** (**`put()`**) and a pure **`recv()`** (**`get()`**). The onus is on the user to ensure synchronization.

Communication costs: ***latency*** and ***bandwidth***.

$$t_t = t_s + Lt_w \tag{5}$$

t_s can include ***software latency*** (cost of negotiating a two-sided transmission, gathering non-contiguous data from memory into a single message).

Note that we have considered t_t as being independent of inter-processor distance (generally well-verified).

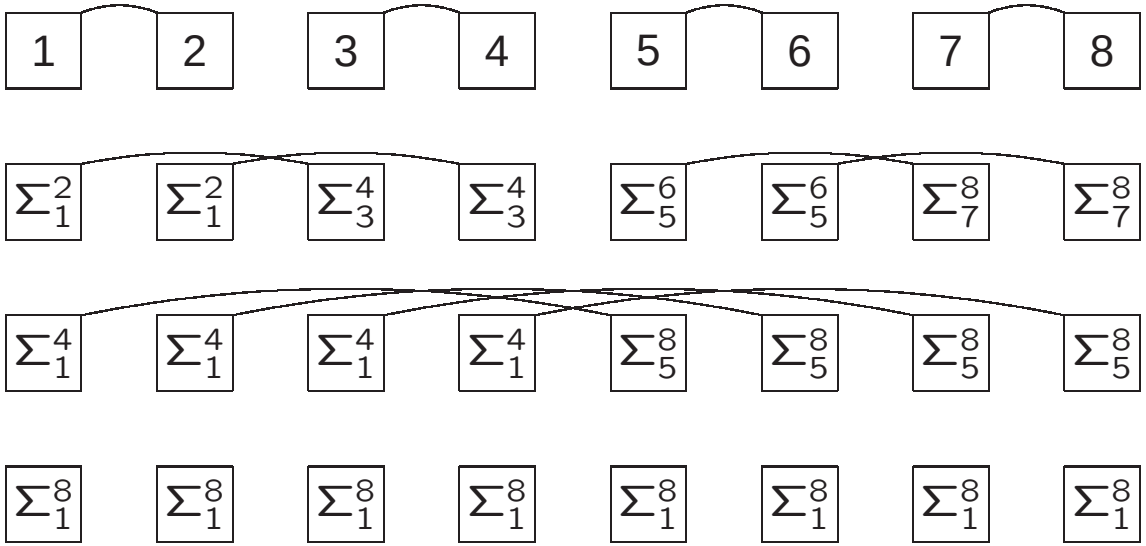
Global reduction

Sum the value of **a** on all PEs, every PE to have a copy of the result. Simplest algorithm: gather on PE 0, sum and broadcast.

```
program test
real :: a, sum
a = my_pe()
call BARRIER()
if( my_pe().EQ.0 )then
    sum = a
    do n = 1,num_pes()-1
       call GET( a, a, 1, n )
       sum = sum + a
    enddo
    do n = 1,num_pes()-1
       call PUT( sum, sum, 1, n )
    enddo
endif
call BARRIER()
print *, 'sum=', sum, ' on PE', my_pe()
end
```

This algorithm on p processors involves $2(p-1)$ communications and p summations, all sequential.

Here's another algorithm for doing the same thing: a binary tree. It executes in $\log_2 p$ steps, each step consisting of one communication and one summation.



There are two ways to perform each step:

```
if( mod(pe,2).EQ.0 )then !execute on even-numbered PEs
    call GET( a, sum, 1, pe+1 )
    sum = sum + a
    call PUT( sum, sum, 1, pe+1 )
endif

if( mod(pe,2).EQ.0 )then !execute on even-numbered PEs
    call GET( a, sum, 1, pe+1 )
    sum = sum + a
else                         !execute on odd-numbered PEs
    call GET( a, sum, 1, pe-1 )
    sum = sum + a
endif
```

The second is faster, even though a redundant computation is performed.

```
do level = 0,lognpes-1    !level on tree
   pedist = 2**level      !distance to sum over
   b(:) = a(:)            !initialize b for each level of the tree
   call barrier()
   if( mod(pe,pedist*2).GE.pedist )then
       call GET( b, c, size(b), pe-pedist, pe-pedist )
       a(:) = b(:) + c(:)
   else
       call GET( b, c, size(b), pe+pedist, pe+pedist )
       a(:) = b(:) + c(:)
   endif
enddo
call barrier()
```

This algorithm performs the summation and distribution in $\log_2 p$ steps.

In general it is better to avoid designating certain PEs for certain tasks. Not only is a better work distribution likely to be available, it can be dangerous code:

```
if( pe.EQ.0 )call barrier()
```

While this is not necessarily incorrect (you could have

```
if( pe.NE.0 )call barrier()
```

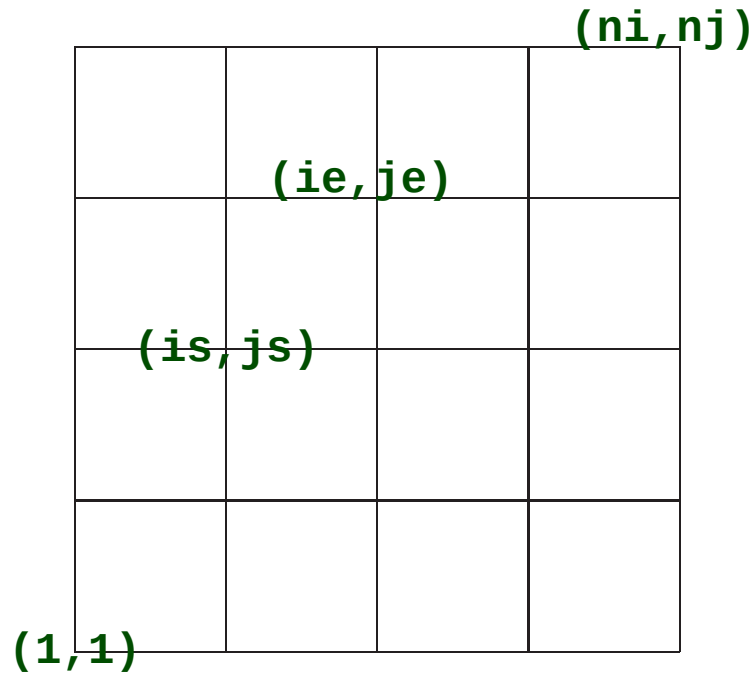
further down in the code), it is easy to go wrong.

Domain decomposition

```
do j = 1,nj
   do i = 1,ni
      a(i,j) = ...
   enddo
enddo
```

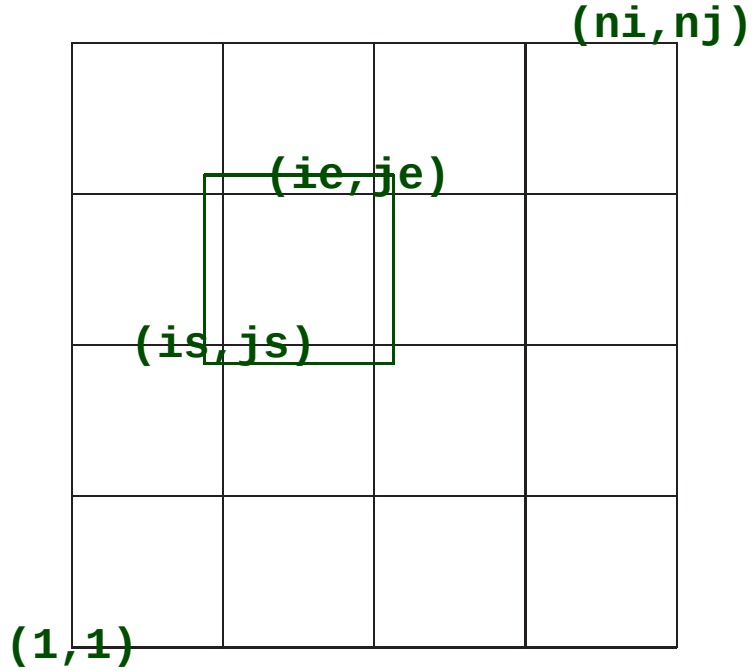
is replaced by

```
do j = js,je
   do i = is,ie
      a(i,j) = ...
   enddo
enddo
```



Computational and data domains

```
do j = js,je
   do i = is.ie
      a(i,j) = ... + a(i-1,j+1)
   end do
end do
```



The **computational domain** is the set of gridpoints that are computed on a domain. The **data domain** is the set of gridpoints needs to be available on-processor to carry out the computation, and includes a **halo** of a certain width.

Diffusion equation

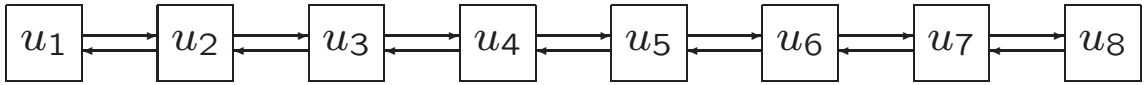
$$\frac{\partial u}{\partial t} - K\frac{\partial^2 u}{\partial x^2} = 0 \tag{6}$$

In discrete form:

$$u_i^{n+1} = u_i^n + K\frac{\Delta t}{\Delta x^2}\left(u_{i+1}^n + u_{i-1}^n - 2u_i^n\right) \tag{7}$$

Assume $P < N$, and that P is an exact divisor of N.

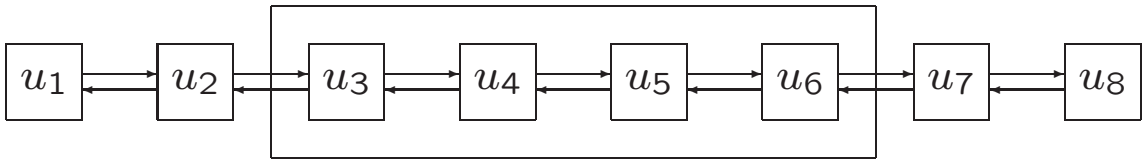
Round-robin allocation



Assign each u_i to a task. Assign each task by rotation to a different processor (***round-robin*** allocation).

```
real :: u(1:N)
do i = 1,N
   if( my_pe().NE.pe(i) )cycle
!pass left,  send u(i) to task i-1, receive u(i+1) from task i+1
   call PUT( u(i), u(i), 1, pe(i-1) )
   call GET( u(i+1), u(i+1), 1, pe(i+1) )
!pass right, send u(i) to task i+1, receive u(i-1) from task i-1
   call PUT( u(i), u(i), 1, pe(i+1) )
   call GET( u(i-1), u(i-1), 1, pe(i-1) )
   call BARRIER()
   u(i) = u(i) + a*( u(i+1)+u(i-1)-2*u(i) )
enddo
```

Block allocation



We could also choose to assign N/P adjacent tasks to the same PE (***block*** allocation).

```
real :: u(l-1:r+1)
!pass left,  send u(l) to task l-1, receive u(r+1) from task r+1
   call PUT( u(l), u(l), 1, pe(l-1) )
   call GET( u(r+1), u(r+1), 1, pe(r+1) )
!pass right, send u(r) to task r+1, receive u(l-1) from task l-1
   call PUT( u(r), u(r), 1, pe(r+1) )
   call GET( u(l-1), u(l-1), 1, pe(l-1) )
   call BARRIER()
   do i = l,r
      u(i) = u(i) + a*( u(i+1)+u(i-1)-2*u(i) )
   enddo
```

Communication is vastly reduced.

Overlapping communication and computation.

```
!pass left,  send u(l) to task l-1, receive u(r+1) from task r+1
   call PUT( u(l), u(l), 1, pe(l-1) )
   call GET( u(r+1), u(r+1), 1, pe(r+1) )
!pass right, send u(r) to task r+1, receive u(l-1) from task l-1
   call PUT( u(r), u(r), 1, pe(r+1) )
   call GET( u(l-1), u(l-1), 1, pe(l-1) )
   do i = l+1,r-1
      u(i) = u(i) + a*( u(i+1)+u(i-1)-2*u(i) )
   enddo
   call BARRIER()
   u(l) = u(l) + a*( u(l+1)+u(l-1)-2*u(l) )
   u(r) = u(r) + a*( u(r+1)+u(r-1)-2*u(r) )
```

The effective communication cost must be measured from the end of the do loop.

Domain decomposition in 2D

There are different ways to partition $N \times N$ points onto P processors.

1D or 2D decomposition?

Assume a problem size $N \times N \times K$, with a halo width of 1.

Cost per timestep with no decomposition:

$$t_0 = N^2 K t_c \tag{8}$$

Cost per timestep with 1D decomposition ($N \times \frac{N}{P} \times K$):

$$t_{1D} = \frac{N^2 K}{P} t_c + 2t_s + 4NKt_w \tag{9}$$

Cost per timestep with 2D decomposition ($\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}} \times K$):

$$t_{2D} = \frac{N^2 K}{P} t_c + 4t_s + \frac{8NK}{\sqrt{P}} t_w \tag{10}$$

In the limit of asymptotic N, P (maintaining constant N^2/P), $t_{2D} \ll t_{1D}$. The case for 2D decomposition is the argument that the communication to computation ratio is like a surface to volume ratio, which goes as $1/r$.

Elliptic equations

Consider a 2D Poisson equation:

$$\nabla^2 u(x, y) = f(x, y) \tag{11}$$

The solution at any point to a boundary value problem in general depends on all other points, therefore incurring a high communication cost on distributed systems.

Jacobi iteration

$$u_{ij}^{(n+1)} = \frac{1}{4}\left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} - \Delta x^2 f_{ij}\right) \qquad (12)$$

Iterate until $|u_{ij}^{(n+1)} - u_{ij}^{(n)}| < \epsilon$.

This method converges under known conditions, but convergence is slow.

Gauss-Seidel iteration

Update values on RHS as they become available:

$$u_{ij}^{(n+1)} = \frac{1}{4}\left(u_{i-1,j}^{(n+1)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n+1)} + u_{i,j+1}^{(n)} - \Delta x^2 f_{ij}\right) \qquad (13)$$

Iterate until $|u_{ij}^{(n+1)} - u_{ij}^{(n)}| < \epsilon$.

This method converges faster, but contains data dependencies that inhibit parallelization.

```
!receive halo from south and west
   call recv(...)
   call recv(...)
!do computation
   do j = js,je
      do i = is,ie
         u(i,j) = u(i-1,j)+u(i,j-1)+u(i+1,j)+u(i,j+1)-a*f(i,j)
      enddo
   enddo
!pass halo to north and east
   call send(...)
   call send(...)
```

4	5	6	7
3	4	5	6
2	3	4	5
1	2	3	4

Red-Black Gauss-Seidel method

```
do parity = red,black
   if( parity.NE.my_parity(pe) )cycle
!receive halo from south and west
   call recv(...)
   call recv(...)
!do red domains on odd passes, black domains on even passes
   do j = js,je
      do i = is,ie
         u(i,j) = u(i-1,j)+u(i,j-1)+u(i+1,j)+u(i,j+1)-a*f(i,j)
      enddo
   enddo
!pass halo to north and east
   call send(...)
   call send(...)
enddo
```

2	1	2	1
1	2	1	2
2	1	2	1
1	2	1	2

More sophisticated methods of hastening the convergence are generally hard to parallelize. The conjugate gradient method accelerates this by computing at each step the optimal vector in phase space along which to converge. Unfortunately, computing the direction involves a global reduction.

In summary, if there are alternatives to solving an elliptic equation over distributed data, they should be given very serious consideration.

Review

- Concurrency and its limitations: Amdahl's law, load imbalance.
- Memory models: shared, distributed, distributed shared.
- Communication primitives: RMA, blocking vs. non-blocking.
- Synchronization: global vs. point-to-point.
- Pitfalls: deadlocks and hangs, race conditions, implicit serialization.
- Algorithms: reducing remote data dependencies.
- Current research: high-level programming models that hide the underlying memory model and configure themselves to the architecture.