Parallel Algorithms

Arnaud Legrand, CNRS, University of Grenoble

LIG laboratory, arnaud.legrand@imag.fr

October 2, 2011

Outline

Part I

Network Models

Motivation

- Scientific computing : large needs in computation or storage resources.
- Need to use systems with “several processors”:
  - Parallel computers with shared/distributed memory
  - Clusters
  - Heterogeneous clusters
  - Clusters of clusters
  - Network of workstations
  - The Grid
  - Desktop Grids
- When modeling platform, *communications modeling* seems to be the most controversial part.
- Two kinds of people produce communication models: those who are concerned with *scheduling* and those who are concerned with *performance evaluation*.
- All these models are *imperfect* and *intractable*.

Outline

UET-UCT

Unit Execution Time - Unit Communication Time.

Hem. . . This one is mainly used by scheduling theoreticians to prove that their problem is hard and to know whether there is some hope to prove some clever result or not.

Some people have introduced a model whith cost of ε for local communications and 1 for communications with the outside.

"Hockney" Model

Hockney [Hoc94] proposed the following model for performance evaluation of the Paragon. A message of size m from P_i to P_j requires:

$$t_{i,j}(m) = L_{i,j} + m/B_{i,j}$$

In scheduling, there are three types of "corresponding" models:

- Communications are not "splitable" and each communication k is associated to a communication time t_k (accounting for message size, latency, bandwidth, middleware, ...).
- Communications are "splitable" but latency is considered to be negligible (linear divisible model):

$$t_{i,j}(m) = m/B_{i,j}$$

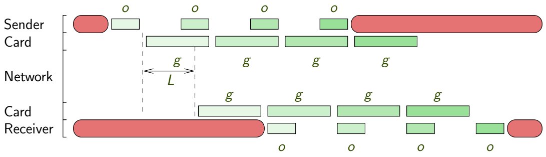
- Communications are "splitable" and latency cannot be neglected (linear divisible model):

$$t_{i,j}(m) = L_{i,j} + m/B_{i,j}$$

LogP

The LogP model [CKP+96] is defined by 4 parameters:

- L is the network latency
- o is the middleware overhead (message splitting and packing, buffer management, connection, ...) for a message of size w
- g is the gap (the minimum time between two packets communication) between two messages of size w
- P is the number of processors/modules

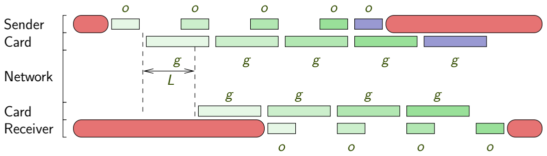


- Sending m bytes with packets of size w:
$$2o + L + \left\lceil \frac{m}{w} \right\rceil \cdot \max(o, g)$$
- Occupation on the sender and on the receiver:
$$o + L + \left(\left\lceil \frac{m}{w} \right\rceil - 1\right) \cdot \max(o, g)$$

LogP

The LogP model [CKP+96] is defined by 4 parameters:

- L is the network latency
- o is the middleware overhead (message splitting and packing, buffer management, connection, ...) for a message of size w
- g is the gap (the minimum time between two packets communication) between two messages of size w
- P is the number of processors/modules



- Sending m bytes with packets of size w:
$$2o + L + \left\lceil \frac{m}{w} \right\rceil \cdot \max(o, g)$$
- Occupation on the sender and on the receiver:
$$o + L + \left(\left\lceil \frac{m}{w} \right\rceil - 1\right) \cdot \max(o, g)$$

LogGP & pLogP

The previous model works fine for short messages. However, many parallel machines have special support for long messages, hence a higher bandwidth. LogGP [AISS97] is an extension of LogP:
G captures the bandwidth for long messages:

short messages $2o + L + \left\lceil \frac{m}{w} \right\rceil \cdot \max(o, g)$

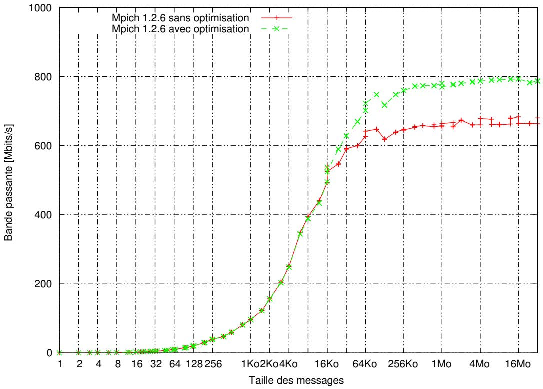
long messages $2o + L + (m-1)G$

There is no fundamental difference...

OK, it works for small and large messages. Does it work for average-size messages ? pLogP [KBV00] is an extension of LogP when L, o and g depends on the message size m. They also have introduced a distinction between o_s and o_r. This is more and more precise but concurency is still not taken into account.

Bandwidth as a Function of Message Size

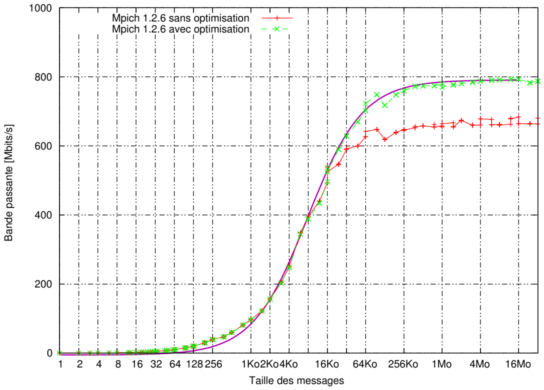
With the Hockney model: $\frac{m}{L+m/B}$.



MPICH, TCP with Gigabit Ethernet

Bandwidth as a Function of Message Size

With the Hockney model: $\frac{m}{L+m/B}$.



MPICH, TCP with Gigabit Ethernet

What About TCP-based Networks?

The previous models work fine for parallel machines. Most networks use TCP that has fancy *flow-control* mechanism and *slow start*. Is it valid to use affine model for such networks?
The answer seems to be yes but latency and bandwidth parameters have to be carefully measured [LQDB05].

- Probing for $m = 1$b and $m = 1$Mb leads to bad results.
- The whole middleware layers should be benchmarked (theoretical latency is useless because of middleware, theoretical bandwidth is useless because of middleware and latency).

The slow-start does not seem to be too harmful.
Most people forget that the round-trip time has a huge impact on the bandwidth.

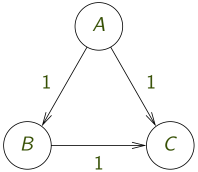
Outline

Multi-ports

- A given processor can communicate with as many other processors as he wishes without any degradation.
- This model is widely used by scheduling theoreticians (think about all DAG with commmunications scheduling problems) to prove that their problem is hard and to know whether there is some hope to prove some clever result or not.

  This model is borderline, especially when allowing duplication, when one communicates with everybody at the same time, or when trying to design algorithms with super tight approximation ratios.

  Frankly, such a model is totally unrealistic.
- Using MPI and synchronous communications, it may not be an issue. However, with multi-core, multi-processor machines, it cannot be ignored...



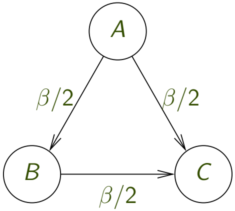
Multi-port

(numbers in s)

Bounded Multi-port

- Assume now that we have threads or multi-core processors.

  We can write that sum of the throughputs of all communications (incomming and outgoing). Such a model is OK for wide-area communications [HP04].
- Remember, the bounds due to the round-trip-time must not be forgotten!

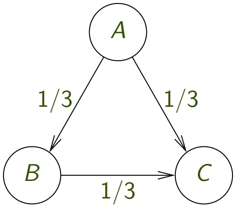


Multi-port (β)

(numbers in Mb/s)

Single-port (Pure)

- A process can communicate with only one other process at a time. This constraint is generally written as a constraint on the sum of communication times and is thus rather easy to use in a scheduling context (even though it complexifies problems).
- This model makes sense when using non-threaded versions of communication libraries (e.g., MPI). As soon as you're allowed to use threads, bounded-multiport seems a more reasonnable option (both for performance and scheduling complexity).

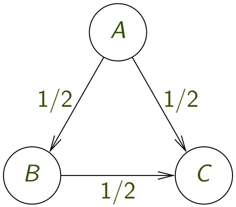


1-port (pure)

(numbers in s)

Single-port (Full-Duplex)

At a given time, a process can be engaged in at most one emission and one reception. This constraint is generally written as two constraints: one on the sum of incomming communication times and one on the sum of outgoing communication times.

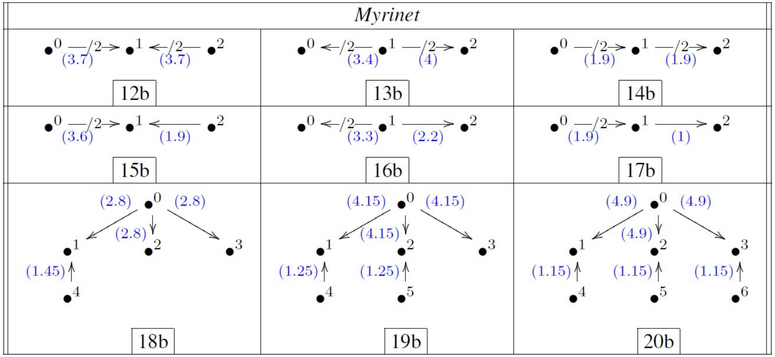


1-port (full duplex)

(numbers in Mb/s)

Single-port (Full-Duplex)

This model somehow makes sense when using networks like Myrinet that have few multiplexing units and with protocols without control flow [Mar07].

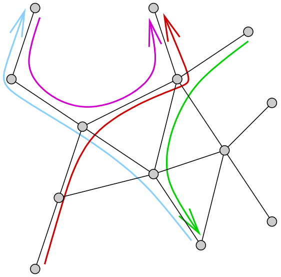


Even if it does not model well complex situations, such a model is not harmfull.

Fluid Modeling

When using TCP-based networks, it is generally reasonnable to use flows to model bandwidth sharing [MR99, Low03].

$$\forall l \in \mathcal{L}, \sum_{r\in\mathcal{R} \text{ s.t. } l\in r} \varrho_r \leqslant c_l$$



Income Maximization maximize $\sum_{r\in\mathcal{R}} \varrho_r$

Max-Min Fairness maximize $\min_{r\in\mathcal{R}} \varrho_r$

Proportional Fairness maximize $\sum_{r\in\mathcal{R}} \log(\varrho_r)$

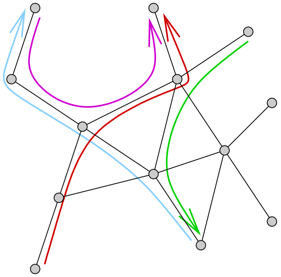
Potential Delay Minimization minimize $\sum_{r\in\mathcal{R}} \frac{1}{\varrho_r}$

Some weird function minimize $\sum_{r\in\mathcal{R}} \arctan(\varrho_r)$

Fluid Modeling

When using TCP-based networks, it is generally reasonnable to use flows to model bandwidth sharing [MR99, Low03].

$$\forall l \in \mathcal{L}, \sum_{r\in\mathcal{R} \text{ s.t. } l\in r} \varrho_r \leqslant c_l$$



Income Maximization maximize $\sum_{r\in\mathcal{R}} \varrho_r$

Max-Min Fairness maximize $\min_{r\in\mathcal{R}} \varrho_r$ ATM

Proportional Fairness maximize $\sum_{r\in\mathcal{R}} \log(\varrho_r)$

TCP Vegas

Potential Delay Minimization minimize $\sum_{r\in\mathcal{R}} \frac{1}{\varrho_r}$

Some weird function minimize $\sum_{r\in\mathcal{R}} \arctan(\varrho_r)$

TCP Reno

Flows Extensions

- Note that this model is a multi-port model with capacity-constraints (like in the previous bounded multi-port).
- When latencies are large, using multiple connections enables to get more bandwidth. As a matter of fact, there is very few to loose in using multiple connections. . .
- Therefore many people enforce a sometimes artificial (but less intrusive) bound on the maximum number of connections per link [Wag05, MYCR06].

Outline

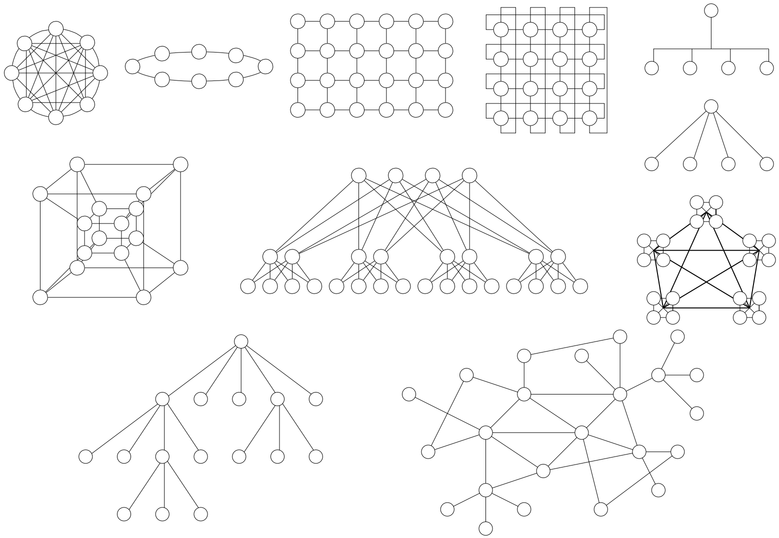
Remind This is a Model, Hence Imperfect

- The previous sharing models are nice but you generally do not know other flows...
- Communications use the memory bus and hence interfere with computations. Taking such interferences into account may become more and more important with multi-core architectures.
- Interference between communications are sometimes...surprising.

Modeling is an art. You have to know your platform and your application to know what is negligeable and what is important. Even if your model is imperfect, you may still derive interesting results.

Outline

Various Topologies Used in the Litterature



Beyond MPI_Comm_rank()?

- So far, MPI gives us a unique number for each processor
- With this one can do anything
- But it's pretty inconvenient because one can do anything with it
- Typically, one likes to impose constraints about which processor/process can talk to which other processor/process
- With this constraint, one can then think of the algorithm in simpler terms
  - There are fewer options for communications between processors
  - So there are fewer choices to implementing an algorithm

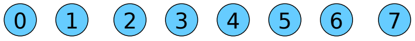
Courtesy of Henri Casanova

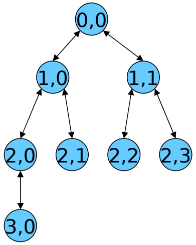
Virtual Topologies?

- MPI provides an abstraction over physical computers
  - Each host has an IP address
  - MPI hides this address with a convenient numbers
  - There could be multiple such numbers mapped to the same IP address
  - All “numbers” can talk to each other
- A Virtual Topology provides an abstraction over MPI
  - Each process has a number, which may be different from the MPI number
  - There are rules about which “numbers” a “number” can talk to
- A virtual topology is defined by specifying the neighbors of each process

Courtesy of Henri Casanova

Implementing a Virtual Topology



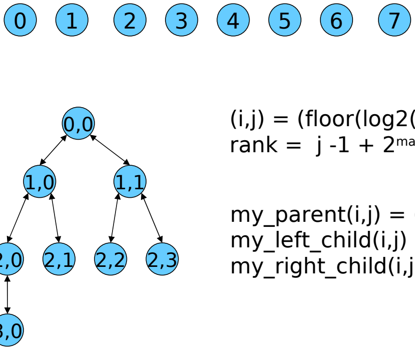


$(i,j) = (\text{floor}(\log2(\text{rank}+1)), \text{rank} - 2^{\max(i,0)}+1)$

$\text{rank} = j - 1 + 2^{\max(i,0)}$

Courtesy of Henri Casanova

Implementing a Virtual Topology

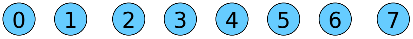


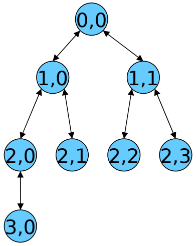
(i,j) = (floor(log2(rank+1)), rank - $2^{\max(i,0)}$+1)
rank = j -1 + $2^{\max(i,0)}$

my_parent(i,j) = (i-1, floor(j/2))
my_left_child(i,j) = (i+1, j*2), if any
my_right_child(i,j) = (i+1, j*2+1), if any

Courtesy of Henri Casanova

Implementing a Virtual Topology





(i,j) = (floor(log2(rank+1)), rank - $2^{\max(i,0)}$+1)
rank = j -1 + $2^{\max(i,0)}$

my_parent(i,j) = (i-1, floor(j/2))
my_left_child(i,j) = (i+1, j*2), if any
my_right_child(i,j) = (i+1, j*2+1), if any

MPI_Send(..., rank(my_parent(i,j)), ...)

MPI_Recv(..., rank(my_left_child(i,j)), ...)

Courtesy of Henri Casanova

Typical Topologies

- Common Topologies (see Section 3.1.2)
  - Linear Array
  - Ring
  - 2-D grid
  - 2-D torus
  - One-level Tree
  - Fully connected graph
  - Arbitrary graph
- Two options for all topologies:
  - Monodirectional links: more constrained but simpler
  - Bidirectional links: less constrained but potential more complicated
    - By “complicated” we typically mean more bug-prone
- We’ll look at Ring and Grid in detail

Courtesy of Henri Casanova

Main Assumption and Big Question

- The main assumption is that once we've defined the virtual topology we forget it's virtual and write parallel algorithms assuming it's physical
  - We assume communications on different (virtual) links do not interfere with each other
  - We assume that computations on different (virtual) processors do not interfere with each other
- The big question: How well do these assumptions hold?
  - The question being mostly about the network
- Two possible "bad" cases
- Case #1: the assumptions do not hold and there are interferences
  - We'll most likely achieve bad performance
  - Our performance models will be broken and reasoning about performance improvements will be difficult
- Case #2: the assumptions do hold but we leave a lot of the network resources unutilized
  - We could perhaps do better with another virtual topology

Courtesy of Henri Casanova

Which Virtual Topology to Pick

- We will see that some topologies are really well suited to certain algorithms
- The question is whether they are well-suite to the underlying architecture
- The goal is to strike a good compromise
  - Not too bad given the algorithm
  - Not too bad given the platform
- Fortunately, many platforms these days use switches, which support naturally many virtual topologies
  - Because they support concurrent communications between disjoint pairs of processors
- As part of a programming assignment, you will explore whether some virtual topology makes sense on our cluster

Courtesy of Henri Casanova

Topologies and Data Distribution

- One of the common steps when writing a parallel algorithm is to distribute some data (array, data structure, etc.) among the processors in the topology
  - Typically, one does data distribution in a way that matches the topology
  - E.g., if the data is 3-D, then it's nice to have a 3-D virtual topology
- One question that arises then is: how is the data distributed across the topology?
- In the next set of slides we look at our first topology: a ring

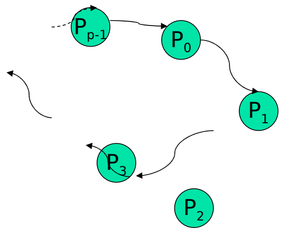
Courtesy of Henri Casanova

Part II

Communications on a Ring

Outline

Ring Topology (Section 3.3)



- Each processor is identified by a rank
  - MY_NUM()
- There is a way to find the total number of processors
  - NUM_PROCS()
- Each processor can send a message to its successor
  - SEND(*addr*, L)
- And receive a message from its predecessor
  - RECV(*addr*, L)

- We'll just use the above pseudo-code rather than MPI
- Note that this is much simpler than the example tree topology we saw in the previous set of slides

Courtesy of Henri Casanova

Virtual vs. Physical Topology

- Now that we have chosen to consider a Ring topology we "pretend" our physical topology is a ring topology
- We can always implement a virtual ring topology (see previous set of slides)
  - And read Section 4.6
- So we can write many "ring algorithms"
- It may be that a better virtual topology is better suited to our physical topology
- But the ring topology makes for very simple programs and is known to be reasonably good in practice
- So it's a good candidate for our first look at parallel algorithms

Courtesy of Henri Casanova

Cost of communication (Sect. 3.2.1)

- It is actually difficult to precisely model the cost of communication
  - E.g., MPI implementations do various optimizations given the message sizes
- We will be using a simple model

  Time = L + m/B

  L: start-up cost or *latency*

  B: bandwidth (b = 1/B)

  m: message size

- We assume that if a message of length m is sent from P_0 to P_q, then the communication cost is q(L + m b)
- There are many assumptions in our model, some not very realistic, but we'll discuss them later

Courtesy of Henri Casanova

Assumptions about Communications

- Several Options
  - Both Send() and Recv() are blocking
    - Called "rendez-vous"
    - Very old-fashioned systems
  - Recv() is blocking, but Send() is not
    - Pretty standard
    - MPI supports it
  - Both Recv() and Send() are non-blocking
    - Pretty standard as well
    - MPI supports it

Courtesy of Henri Casanova

Assumptions about Concurrency

- One question that's important is: can the processor do multiple things at the same time?
- Typically we will assume that the processor can send, receive, **and** compute at the same time
  - Call MPI_IRecv() Call MPI_ISend()
  - Compute something
- This of course implies that the three operations are independent
  - E.g., you don't want to send the result of the computation
  - E.g., you don't want to send what you're receiving (forwarding)
- When writing parallel algorithms (in pseudo-code), we'll simply indicate concurrent activities with a || sign

Courtesy of Henri Casanova

Collective Communications

- To write a parallel algorithm, we will need collective operations
  - Broadcasts, etc.
- Now MPI provide those, and they likely:
  - Do not use the ring logical topology
  - Utilize the physical resources well
- Let's still go through the exercise of writing some collective communication algorithms
- We will see that for some algorithms we really want to do these communications "by hand" on our virtual topology rather than using the MPI collective

Courtesy of Henri Casanova

Outline

Broadcast (Section 3.3.1)

- We want to write a program that has P_k send the same message of length m to all other processors

  Broadcast(k,*addr*,m)
- On the ring, we just send to the next processor, and so on, with no parallel communications whatsoever
- This is of course not the way one should implement a broadcast in practice if the physical topology is not merely a ring
  - MPI uses some type of tree topology

Courtesy of Henri Casanova

Broadcast (Section 3.3.1)

```
Brodcast(k,addr,m)
  q = MY_NUM()
  p = NUM_PROCS()
  if (q == k)
     SEND(addr,m)
  else
     if (q == k-1 mod p)
        RECV(addr,m)
     else
        RECV(addr,m)
        SEND(addr,m)
     endif
  endif
```

- Assumes a blocking receive
- Sending may be non-blocking
- The broadcast time is

$(p-1)(L+m\ b)$

Outline

Scatter (Section 3.2.2)

- Processor k sends a different message to all other processors (and to itself)
  - P_k stores the message destined to P_q at address addr[q], including a message at addr[k]
- At the end of the execution, each processor holds the message it had received in msg
- The principle is just to pipeline communication by starting to send the message destined to P_{k-1}, the most distant processor

Courtesy of Henri Casanova

Scatter (Section 3.3.2)

```
Scatter(k,msg,addr,m)
  q = MY_NUM()
  p = NUM_PROCS()
  if (q == k)
    for i = 0 to p-2
        SEND(addr[k+p-1-i mod p],m)
    msg ← addr[k]
  else
    RECV(tempR,L)
    for i = 1 to k-1-q mod p
        tempS ↔ tempR
        SEND(tempS,m) || RECV(tempR,m)
    msg ← tempR
```

Same execution time as the broadcast
$(p-1)(L + m\,b)$

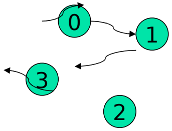
Swapping of send buffer and receive buffer (pointer)

Sending and Receiving in Parallel, with a non blocking Send

Courtesy of Henri Casanova

Scatter (Section 3.3.2)

```
Scatter(k,msg,addr,m)
  q = MY_NUM()
  p = NUM_PROCS()
  if (q == k)
     for i = 0 to p-2
        SEND(addr[k+p-1-i mod p],m)
     msg ← addr[k]
  else
     RECV(tempR,L)
     for i = 1 to k-1-q mod p
        tempS ↔ tempR
        SEND(tempS,m)  || RECV(tempR,m)
     msg ← tempR
```



k = 2, p = 4

Proc q=2
send addr[2+4-1-0 % 4 = 1]
send addr[2+4-1-1 % 4 = 0]
send addr[2+4-1-2 % 4 = 3]
msg = addr[2]

Proc q=3
recv (addr[1])
// loop 2-1-3 % 4 = 2 times
send (addr[1]) || recv (addr[0])
send (addr[0]) || recv (addr[3])

msg = addr[3]

Proc q=0
recv (addr[1])
// loop 2-1-2 % 4 = 1 time
send (addr[1]) || recv (addr[0])

msg = addr[0]

Proc q=1
// loop 2-1-1 % 4 = 0 time
recv (addr[1])

msg = addr[1]

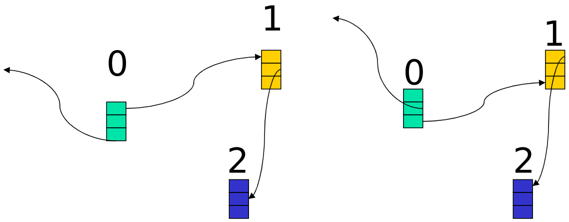
Courtesy of Henri Casanova

Outline

All-to-all (Section 3.3.3)

```
All2All(my_addr, addr, m)
  q = MY_NUM()
  p = NUM_PROCS()
  addr[q] ← my_addr
  for i = 1 to p-1
      SEND(addr[q-i+1 mod p],m)
   || RECV(addr[q-i mod p],m)
```

Same execution time as the scatter
(p-1)(L + m b)



Courtesy of Henri Casanova

Outline

A faster broadcast?

- How can we improve performance?
- One can cut the message in many small pieces, say in r pieces where m is divisible by r.
- The root processor just sends r messages
- The performance is as follows
  - Consider the last processor to get the last piece of the message
  - There need to be p-1 steps for the first piece to arrive, which takes $(p-1)(L + m b / r)$
  - Then the remaining r-1 pieces arrive one after another, which takes $(r-1)(L + m b / r)$
  - For a total of: $(p - 2 + r) (L + mb / r)$

Courtesy of Henri Casanova

A faster broadcast?

- The question is, what is the value of r that minimizes

  (p - 2 + r) (L + m b / r) ?

- One can view the above expression as (c+ar)(d+b/r), with four constants a, b, c, d
- The non-constant part of the expression is then ad.r + cb/r, which must be minimized
- It is known that this value is minimized for

  sqrt(cb / ad)

  and we have

  r_{opt} = sqrt(m(p-2) b / L)

  with the optimal time

  (sqrt((p-2) L) + sqrt(m b))^2

  which tends to mb when m is large, which is independent of p!

Courtesy of Henri Casanova

Well-known Network Principle

- We have seen that if we cut a (large) message in many (small) messages, then we can send the message over multiple hops (in our case p-1) almost as fast as we can send it over a single hop
- This is a fundamental principle of IP networks
  - We cut messages into IP frames
  - Send them over many routers
  - But really go as fast as the slowest router

Courtesy of Henri Casanova

Part III

Speedup and Efficiency

Outline

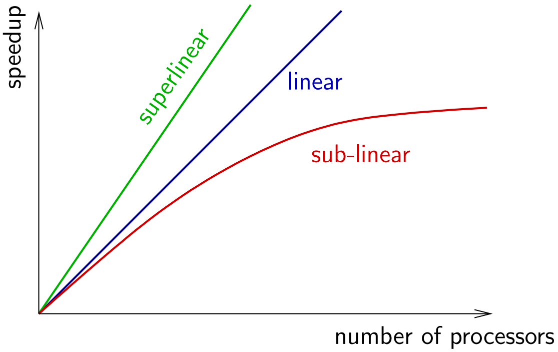
Speedup

- We need a metric to quantify the impact of your performance enhancement
- Speedup: ratio of “old” time to “new” time
  - new time = 1h
  - speedup = 2h / 1h = 2
- Sometimes one talks about a “slowdown” in case the “enhancement” is not beneficial
  - Happens more often than one thinks

Parallel Performance

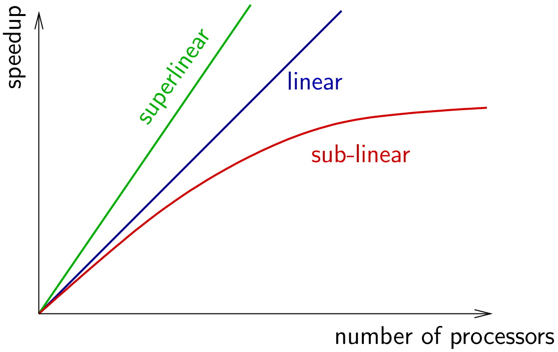
- The notion of speedup is completely generic
  - By using a rice cooker I've achieved a 1.20 speedup for rice cooking
- For parallel programs one defines the Parallel Speedup (we'll just say "speedup"):
  - Parallel program takes time T_1 on 1 processor
  - Parallel program takes time T_p on p processors
  - Parallel Speedup: $S(p) = \frac{T_1}{T_p}$
- In the ideal case, if my sequential program takes 2 hours on 1 processor, it takes 1 hour on 2 processors: called linear speedup

Speedup



Superlinear Speedup? There are several possible causes

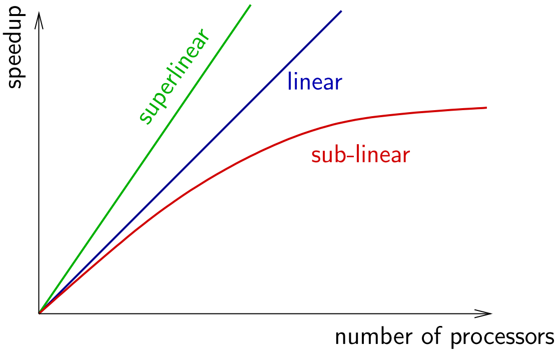
Speedup



Superlinear Speedup? There are several possible causes

Algorithm with optimization problems, throwing many processors at it increases the chances that one will “get lucky” and find the optimum fast

Speedup



Superlinear Speedup? There are several possible causes

Algorithm with optimization problems, throwing many processors at it increases the chances that one will "get lucky" and find the optimum fast

Hardware with many processors, it is possible that the entire application data resides in cache (vs. RAM) or in RAM (vs. Disk)

Outline

Bad News: Amdahl's Law

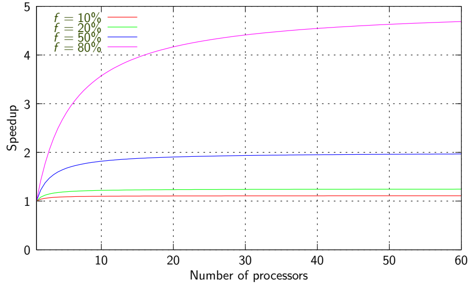
Consider a program whose execution consists of two phases

1. One sequential phase : $T_{seq} = (1 - f)T_1$
2. One phase that can be perfectly parallelized (linear speedup) $T_{par} = fT_1$

Therefore: $T_p = T_{seq} + T_{par}/p = (1 - f)T_1 + fT_1/p$.

Amdahl's Law:

$$S_p = \frac{1}{1 - f + \frac{f}{p}}$$



Lessons from Amdahl's Law

- It's a law of diminishing return
- If a significant fraction of the code (in terms of time spent in it) is not parallelizable, then parallelization is not going to be good
- It sounds obvious, but people new to high performance computing often forget how bad Amdahl's law can be
- Luckily, many applications can be almost entirely parallelized and f is small

Parallel Efficiency

- Efficiency is defined as $\mathit{Eff}(p) = S(p)/p$
- Typically < 1, unless linear or superlinear speedup
- Used to measure how well the processors are utilized
  - If increasing the number of processors by a factor 10 increases the speedup by a factor 2, perhaps it's not worth it: efficiency drops by a factor 5
  - Important when purchasing a parallel machine for instance: if due to the application's behavior efficiency is low, forget buying a large cluster

Scalability

- Measure of the "effort" needed to maintain efficiency while adding processors
- Efficiency also depends on the problem size: $\mathit{Eff}(n, p)$
- Isoefficiency: At which rate does the problem size need to be increase to maintain efficiency
  - $n_c(p)$ such that $\mathit{Eff}(n_c(p), p) = c$
  - By making a problem ridiculously large, on can typically achieve good efficiency
  - Problem: is it how the machine/code will be used?

Part IV

Algorithms on a Ring

Outline

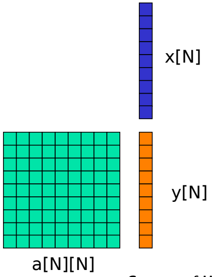
Parallel Matrix-Vector product

- y = A x
- Let n be the size of the matrix

```
int a[n][n];
int x[n];
for i = 0 to n-1 {
  y[i] = 0;
  for j = 0 to n-1
    y[i] = y[i] + a[i,j] * x[j];
}
```

- How do we do this in parallel?

Section 4.1 in the book

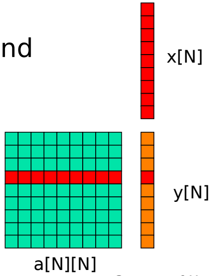


Courtesy of Henri Casanova

Parallel Matrix-Vector product

- How do we do this in parallel?
- For example:
  - Computations of elements of vector y are independent
  - Each of these computations requires one row of matrix a and vector x
- In shared-memory:

```
#pragma omp parallel for private(i,j)
for i = 0 to n-1 {
   y[i] = 0;
   for j = 0 to n-1
      y[i] = y[i] + a[i,j] * x[j];
}
```



Courtesy of Henri Casanova

Parallel Matrix-Vector Product

- In distributed memory, one possibility is that each process has a full copy of matrix a and of vector x
- Each processor declares a vector y of size n/p
  - We assume that p divides n
- Therefore, the code can just be

```
load(a); load(x)
p = NUM_PROCS(); r = MY_RANK();
for (i=r*n/p; i<(r+1)*n/p; i++) {
      for (j=0;j<n;j++)
      y[i-r*n/p] = a[i][j] * x[j];
}
```

- It's embarrassingly parallel
- What about the result?

Courtesy of Henri Casanova

What about the result?

- After the processes complete the computation, each process has a piece of the result
- One probably wants to, say, write the result to a file
  - Requires synchronization so that the I/O is done correctly
- For example

```
 . . .
if (r != 0) {
      recv(&token,1);
}
open(file, "append");
for (j=0; j<n/p ; j++)
      write(file, y[j]);
send(&token,1);
close(file)
barrier();   // optional
```

- Could also use a "gather" so that the entire vector is returned to processor 0
  - vector y fits in the memory of a single node

Courtesy of Henri Casanova

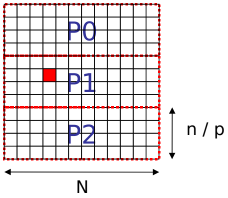
What if matrix a is too big?

- Matrix a may not fit in memory
  - Which is a motivation to use distributed memory implementations
- In this case, each processor can store only a piece of matrix a
- For the matrix-vector multiply, each processor can just store n/p rows of the matrix
  - Conceptually: A[n][n]
  - But the program declares a[n/p][n]
- This raises the (annoying) issue of global indices versus local indices

Courtesy of Henri Casanova

Global vs. Local indices

- When an array is split among processes
  - global index (I,J) that references an element of the matrix
  - local index (i,j) that references an element of the local array that stores a piece of the matrix
  - Translation between global and local indices
    - think of the algorithm in terms of global indices
    - implement it in terms of local indices



Global: A[5][3]
Local: a[1][3] on process P1

a[i,j] = A[(n/p)*rank + i][j]

Courtesy of Henri Casanova

Global Index Computation

- Real-world parallel code often implements actual translation functions
  - GlobalToLocal()
  - LocalToGlobal()
- This may be a good idea in your code, although for the ring topology the computation is pretty easy, and writing functions may be overkill
- We'll see more complex topologies with more complex associated data distributions and then it's probably better to implement such functions

Courtesy of Henri Casanova

Distributions of arrays

- At this point we have
  - 2-D array a distributed
  - 1-D array y distributed
  - 1-D array x replicated
- Having distributed arrays makes it possible to partition work among processes
  - But it makes the code more complex due to global/local indices translations
  - It may require synchronization to load/save the array elements to file

Courtesy of Henri Casanova

All vector distributed?

- So far we have array x replicated
- It is usual to try to have all arrays involved in the same computation be distributed in the same way
  - makes it easier to read the code without constantly keeping track of what's distributed and what's not
    - e.g., "local indices for array y are different from the global ones, but local indices for array x are the same as the global ones" will lead to bugs
- What one would like it for each process to have
  - N/n rows of matrix A in an array a[n/p][n]
  - N/n components of vector x in an array x[n/p]
  - N/n components of vector y in an array y[n/p]
- Turns out there is an elegant solution to do this

Courtesy of Henri Casanova

Principle of the Algorithm

$$P_0 \quad \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} \quad \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$P_1 \quad \begin{pmatrix} A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \end{pmatrix} \quad \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

$$P_2 \quad \begin{pmatrix} A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \end{pmatrix} \quad \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}$$

$$P_3 \quad \begin{pmatrix} A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} \quad \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}$$

Initial data distribution for:

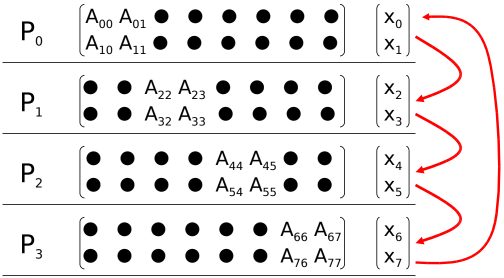
$n = 8$

$p = 4$

$n/p = 2$

Courtesy of Henri Casanova

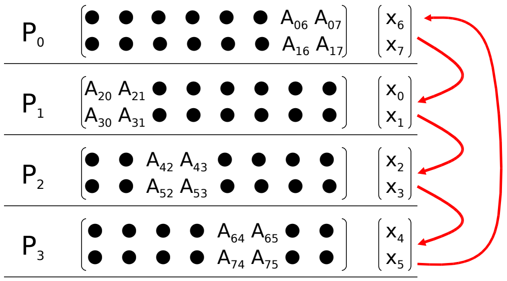
Principle of the Algorithm



Step 0

Courtesy of Henri Casanova

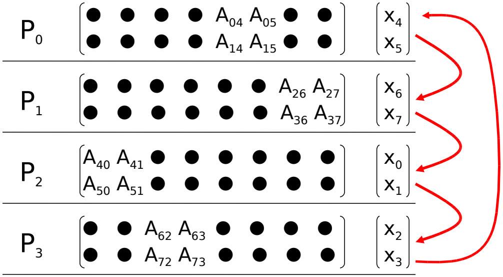
Principle of the Algorithm



Step 1

Courtesy of Henri Casanova

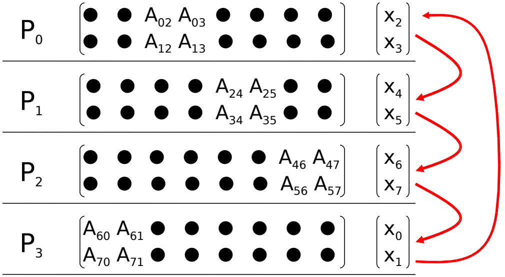
Principle of the Algorithm



Step 2

Courtesy of Henri Casanova

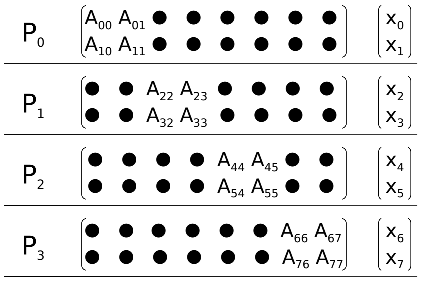
Principle of the Algorithm



Step 3

Courtesy of Henri Casanova

Principle of the Algorithm



The final exchange of vector x is not strictly necessary, but one may want to have it distributed as the end of the computation like it was distributed at the beginning.

Final state

Courtesy of Henri Casanova

Algorithm

- Uses two buffers
  - tempS for sending and tempR to receiving

```
float A[n/p][n], x[n/p], y[n/p];
r ← n/p
tempS ← x   /* My piece of the vector (n/p elements) */
for (step=0; step<p; step++) {  /* p steps */
  SEND(tempS,r)
  RECV(tempR,r)
  for (i=0; i<n/p; i++)
    for (j=0; j <n/p; j++)
      y[i] ← y[i] + a[i,(rank - step mod p) * n/p + j] * tempS[j]
  tempS ↔ tempR
}
```

- In our example, process of rank 2 at step 3 would work with the 2x2 matrix block starting at column ((2 - 3) mod 4)*8/4 = 3 * 8 / 4 = 6;

Courtesy of Henri Casanova

A few General Principles

- Large data needs to be distributed among processes (running on different nodes of a cluster for instance)
  - causes many arithmetic expressions for index computation
  - People who do this for a leaving always end up writing local_to_global() and global_to_local() functions
- Data may need to be loaded/written before/after the computation
  - requires some type of synchronization among processes
- Typically a good idea to have all data structures distributed similarly to avoid confusion about which indices are global and which ones are local
  - In our case, all indices are local
- In the end the code looks much more complex than the equivalent OpenMP implementation

Courtesy of Henri Casanova

Performance

- There are p identical steps
- During each step each processor performs three activities: computation, receive, and sending
  - Computation: r^2 w
    - w: time to perform one += * operation
  - Receiving: L + r b
  - Sending: L + r b

$$T(p) = p\,(r^2w + 2L + 2rb)$$

Courtesy of Henri Casanova

Asymptotic Performance

- $T(p) = p(r^2w + 2L + 2rb)$
  - $\text{Speedup}(p) = n^2w \ / \ p\,(r^2w + 2L + 2rb)$
    $= n^2w \ / \ (n^2w/p + 2pL + 2nb)$
  - $\text{Eff}(p) = n^2w \ / \ (n^2w + 2p^2L + 2pnb)$
  - For p fixed, when n is large, $\text{Eff}(p) \sim 1$
- Conclusion: the algorithm is asymptotically optimal

Courtesy of Henri Casanova

Performance (2)

- Note that an algorithm that initially broadcasts the entire vector to all processors and then have every processor compute independently would be in time

  $$(p-1)(L + n\,b) + pr^2 w$$

  - Could use the pipelined broadcast
- which:
  - has the same asymptotic performance
  - is a simpler algorithm
  - wastes only a tiny little bit of memory
  - is arguably much less elegant
- It is important to think of simple solutions and see what works best given expected matrix size, etc.

Courtesy of Henri Casanova

Back to the Algorithm

```
float A[n/p][n], x[n/p], y[n/p];
r ← n/p
tempS ← x   /* My piece of the vector (n/p elements) */
for (step=0; step<p; step++) {  /* p steps */
  SEND(tempS,r)
  RECV(tempR,r)
  for (i=0; i<n/p; i++)
    for (j=0; j <n/p; j++)
      y[i] ← y[i] + a[i,(rank - step mod p) * n/p + j] * tempS[j]
  tempS ↔ tempR
}
```

- In the above code, at each iteration, the SEND, the RECV, and the computation can all be done in parallel
- Therefore, one can overlap communication and computation by using non-blocking SEND and RECV if available
- MPI provides MPI_ISend() and MPI_IRecv() for this purpose

Courtesy of Henri Casanova

Nore Concurrent Algorithm

- Notation for concurrent activities:

```
float A[n/p][n], x[n/p], y[n/p];
tempS ← x   /* My piece of the vector (n/p elements) */
r ← n/p
for (step=0; step<p; step++) {  /* p steps */
     SEND(tempS,r)
  || RECV(tempR,r)
  || for (i=0; i<n/p; i++)
       for (j=0; j <n/p; j++)
         y[i] ← y[i]+a[i,(rank-step mod p)*n/p+j]*tempS[j]
  tempS ↔ tempR
}
```

Courtesy of Henri Casanova

Better Performance

- There are p identical steps
- During each step each processor performs three activities: computation, receive, and sending
  - Computation: r^2w
  - Receiving: L + rb
  - Sending: L + rb

$$T(p) = p\ \max(r^2w\ ,\ L + rb)$$

Same asymptotic performance as above, but better performance for smaller values of n

Courtesy of Henri Casanova

Hybrid parallelism

- We have said many times that multi-core architectures are about to become the standard
- When building a cluster, the nodes you will buy will be multi-core
- Question: how to exploit the multiple cores?
  - Or in our case how to exploit the multiple processors in each node
- Option #1: Run multiple processes per node
  - Causes more overhead and more communication
  - In fact will cause network communication among processes within a node!
    - MPI will not know that processes are co-located

Courtesy of Henri Casanova

OpenMP MPI Program

- Option #2: Run a single multi-threaded process per node
  - Much lower overhead, fast communication within a node
  - Done by combining MPI with OpenMP!
- Just write your MPI program
- Add OpenMP pragmas around loops
- Let's look back at our Matrix-Vector multiplication example

Courtesy of Henri Casanova

Hybrid Parallelism

```
float A[n/p][n], x[n/p], y[n/p];
tempS ← x   /* My piece of the vector (n/p elements) */
for (step=0; step<p; step++) {  /* p steps */
     SEND(tempS,r)
  || RECV(tempR,r)
  || #pragma omp parallel for private(i,j)
     for (i=0; i<n/p; i++)
       for (j=0; j <n/p; j++)
         y[i] ← y[i] + a[i,(rank - step mod p)*n/p+j]*
                          tempS[j]
  tempS ↔ tempR
}
```

- This is called Hybrid Parallelism
- Communication via the network among nodes
- Communication via the shared memory within nodes

Courtesy of Henri Casanova

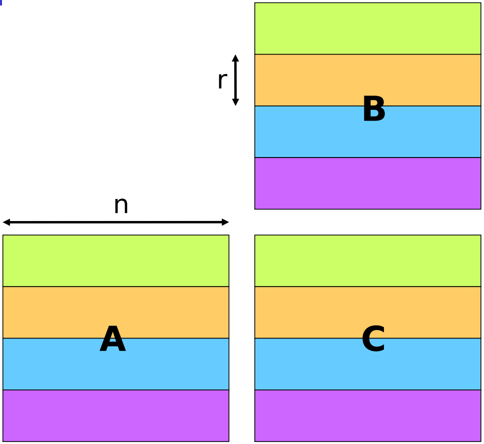
Outline

Matrix Multiplication on the Ring

- See Section 4.2
- Turns out one can do matrix multiplication in a way very similar to matrix-vector multiplication
  - A matrix multiplication is just the computation of n^2 scalar products, not just n
- We have three matrices, A, B, and C
- We want to compute C = A*B
- We distribute the matrices to that each processor "owns" a block row of each matrix
  - Easy to do if row-major is used because all matrix elements owned by a processor are contiguous in memory

Courtesy of Henri Casanova

Data Distribution

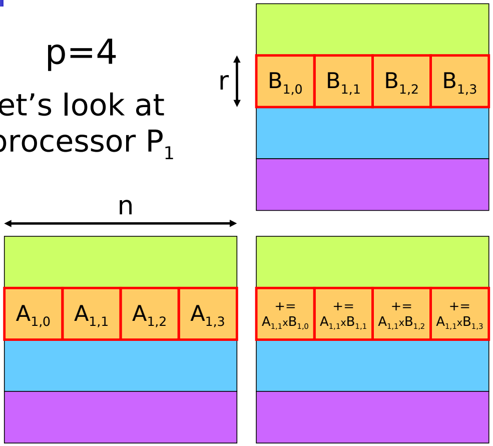


Courtesy of Henri Casanova

First Step

p=4

let's look at processor P_1

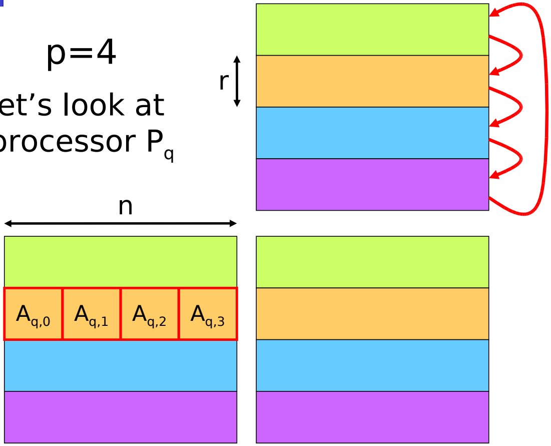

r
$B_{1,0}$
$B_{1,1}$
$B_{1,2}$
$B_{1,3}$
n
$A_{1,0}$
$A_{1,1}$
$A_{1,2}$
$A_{1,3}$
$+= A_{1,1} \times B_{1,0}$
$+= A_{1,1} \times B_{1,1}$
$+= A_{1,1} \times B_{1,2}$
$+= A_{1,1} \times B_{1,3}$

Courtesy of Henri Casanova

Shifting of block rows of B

p=4

let's look at processor P_q

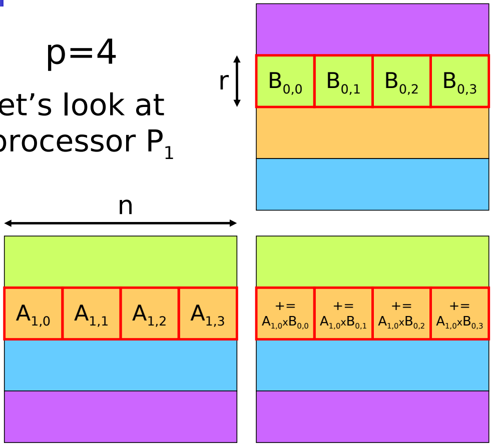


Courtesy of Henri Casanova

Second step

p=4

let’s look at processor P_1


r
$B_{0,0}$
$B_{0,1}$
$B_{0,2}$
$B_{0,3}$
n
$A_{1,0}$
$A_{1,1}$
$A_{1,2}$
$A_{1,3}$
$+= A_{1,0} \times B_{0,0}$
$+= A_{1,0} \times B_{0,1}$
$+= A_{1,0} \times B_{0,2}$
$+= A_{1,0} \times B_{0,3}$

Courtesy of Henri Casanova

Algorithm

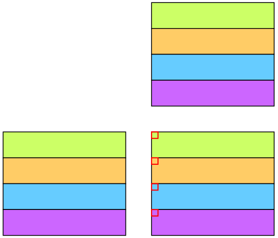
- In the end, every Ci,j block has the correct value: $A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \ldots$
- Basically, this is the same algorithm as for matrix-vector multiplication, replacing the partial scalar products by submatrix products (gets tricky with loops and indices)

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
  tempS ↔ tempR
}
```

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank − step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

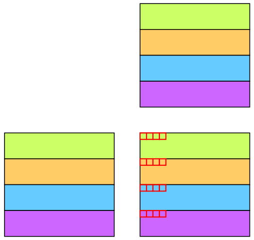


step=0
l=0
i=0
j=0

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank − step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

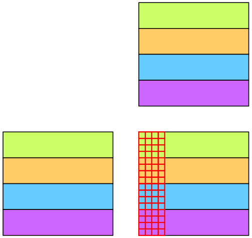


step=0
l=0
i=0
j=*

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

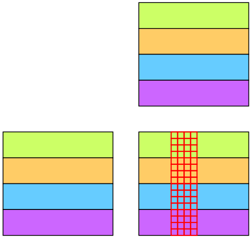


step=0
l=0
i=*
j=*

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
      for (i=0; i<N/p; i++)
       for (j=0; j<N/p; j++)
        for (k=0; k<N/p; k++)
          C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

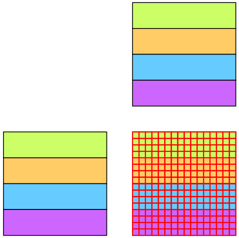


step=0
l=1
i=*
j=*

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
     SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
      for (i=0; i<N/p; i++)
       for (j=0; j<N/p; j++)
        for (k=0; k<N/p; k++)
          C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

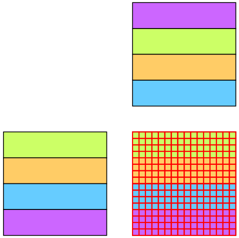


step=0
l=*
i=*
j=*

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

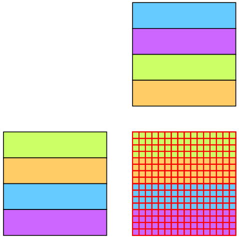


step=1
l=*
i=*
j=*

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```

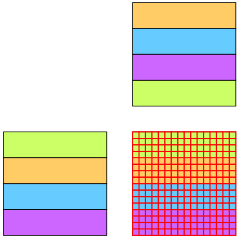


step=2
l=*
i=*
j=*

Courtesy of Henri Casanova

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
      for (i=0; i<N/p; i++)
       for (j=0; j<N/p; j++)
        for (k=0; k<N/p; k++)
          C[i,lr+j] ← C[i,lr+j] + A[i,r((rank − step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=3
l=*
i=*
j=*

Courtesy of Henri Casanova

Performance

- Performance Analysis is straightforward
- p steps and each step takes time:

  max (nr^2 w, L + nrb)
  - p rxr matrix products = pr^3 = nr^2 operations
- Hence, the running time is:

  T(p) = p max (nr^2 w , L + nrb)
- Note that a naive algorithm computing n Matrix-vector products in sequence using our previous algorithm would take time

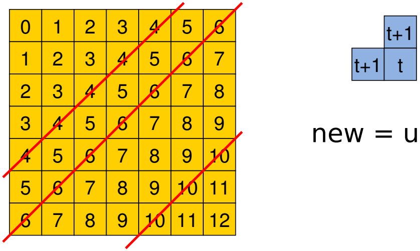
  T(p) = p max(nr^2 w , **n**L + nrb)
- We just saved network latencies!

Courtesy of Henri Casanova

Outline

Stencil Application (Section 4.3)

- We've talked about stencil applications in the context of shared-memory programs



new = update(old,W,N)

- We found that we had to cut the matrix in "small" blocks
  - On a ring the same basic idea applies, but let's do it step-by-step

Courtesy of Henri Casanova

Stencil Application

- Let us, for now, consider that the domain is of size nxn and that we have p=n processors
  - Classic way to first approach a problem
- Each processor is responsible for computing one row of the domain (at each iteration)
- Each processor holds one row of the domain and has the following declaration:

```
var   A: array[0..n-1] of real
```

- One first simple idea is to have each processor send each cell value to its neighbor as soon as that cell value is computed
- Basic principle: do communication as early as possible to get your "neighbors" started as early as possible
  - Remember that one of the goals of a parallel program is to reduce idle time on the processors
- We call this algorithm the Greedy algorithm, and seek an evaluation of its performance

Courtesy of Henri Casanova

The Greedy Algorithm

```
q = MY_NUM()
p = NUM_PROCS
if (q == 0) then
    A[0] = Update(A[0],nil,nil)
    Send(A[0],1)
else
    Recv(v,1)
    A[0] = Update(A[0],nil,v)
endif
for j = 1 to n-1
    if (q == 0) then
        A[j] = Update(A[j], A[j-1], nil)
        Send(A[j],1)
    elsif (q == p-1) then
        Recv(v,1)
        A[j] = Update(A[j], A[j-1], v)
    else
        Send(A[j-1], 1)  ||  Recv(v,1)
        A[j] = Update(A[j], A[j-1], v)
    endif
endfor
```

First element of the row

Other elements

note the use of “nil”
for borders and corners

Courtesy of Henri Casanova

Greedy Algorithm

- This is all well and good, but typically we have n > p
- Assuming that p divides n, each processor will hold n/p rows
  - Good for load balancing
- The goal of a greedy algorithm is always to allow processors to start computing as early as possible
- This suggests a cyclic allocation of rows among processors

P0
P1
P2
P0
P1
P2
P0
P1
P2

- P1 can start computing after P0 has computed its first cell

Courtesy of Henri Casanova

Greedy Algorithm

- Each processor holds n/p rows of the domain
- Thus it declares:

  var A[0..n/p-1,n] of real
- Which is a contiguous array of rows, with these rows not contiguous in the domain
  - Therefore we have a non-trivial mapping between global indices and local indices, but we'll see that they don't appear in the code
- Let us rewrite the algorithm

Courtesy of Henri Casanova

The Greedy Algorithm

```
p = MY_NUM()
q = NUM_PROCS
For i = 0 to n/p -1
    if (q == 0) and (i == 0) then
          A[0,0] = Update(A[0,0],nil,nil)
          Send(A[0],1)
    else
          Recv(v,1)
          A[i,0] = Update(A[i,0],nil,v)
    endif
    for j = 1 to n-1
          if (q == 0) and (i == 0) then
                    A[i,j] = Update(A[i,j], A[i,j-1], nil)
                    Send(A[i,j],1)
          elsif (q == p-1) and (i = n/p-1) then
                    Recv(v,1)
                    A[i,j] = Update(A[i,j], A[i-1,j], v)
          else
                    Send(A[i,j-1], 1)   ||   Recv(v,1)
                    A[i,j] = Update(A[i,j], A[i-1,j-1], v)
          endif
    endfor
endfor
```

Courtesy of Henri Casanova

Performance Analysis

- Let T(n,p) denote the computation time of the algorithm for a nxn domain and with p processors
- A each step a processor does at most three things
  - Receive a cell
  - Send a cell
  - Update a cell
- The algorithm is “clever” because at each step k, the sending of messages from step k is overlapped with the receiving of messages at step k+1
- Therefore, the time needed to compute one algorithm step is the sum of
  - Time to send/receive a cell: L + b
  - Time to perform a cell update: w
- So, if we can count the number of steps, we can simply multiply and get the overall execution time

Courtesy of Henri Casanova

Performance Analysis

- It takes p-1 steps before processor P_{p-1} can start computing its first cell
- Thereafter, this processor can compute one cell at every step
- The processor holds n*n/p cells
- Therefore, the whole program takes: p-1+n*n/p steps
- And the overall execution time:
  $T(n,p) = (p - 1 + n^2/p)(w + L + b)$
- The sequential time is: n^2w
- The Speedup, $S(n,p) = n^2w / T(n,p)$
- When n gets large, $T(n,p) \sim n^2/p\,(w + L + b)$
- Therefore, $Eff(n,p) \sim w / (w + L + b)$
- This could be WAY below one
  - In practice, and often, $L + b >> w$
- Therefore, this greedy algorithm is probably not a good idea at all!

Courtesy of Henri Casanova

Granularity

- How do we improve on performance?
- What really kills performance is that we have to do so much communication
  - Many bytes of data
  - Many individual messages
- So we we want is to augment the granularity of the algorithm
  - Our “tasks” are not going to be “update one cell” but instead “update multiple cells”
- This will allow us to reduce both the amount of data communicated and the number of messages exchanged

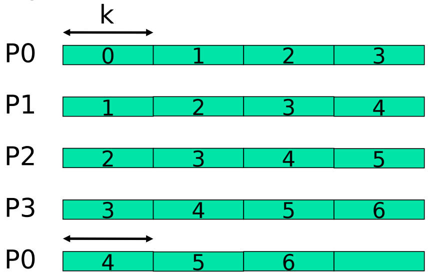
Courtesy of Henri Casanova

Reducing the Granularity

- A simple approach: have a processor compute k cells in sequence before sending them
- This is in conflict with the “get processors to compute as early as possible” principle we based our initial greedy algorithm on
  - So we will reduce communication cost, but will increase idle time
- Let use assume that k divides n
- Each row now consists of n/k segments
  - If k does not divide n we have left over cells and it complicates the program and the performance analysis and as usual doesn’t change the asymptotic performance analysis

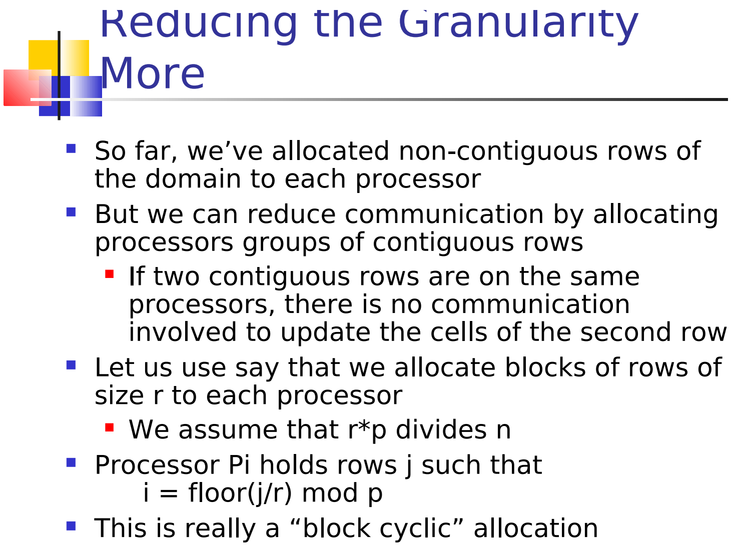
Courtesy of Henri Casanova

Reducing the Granularity



- The algorithm computes segment after segment
- The time before P1 can start computing is the time for P0 to compute a whole segment
- Therefore, it will take longer until P_{p-1} can start computing

Courtesy of Henri Casanova

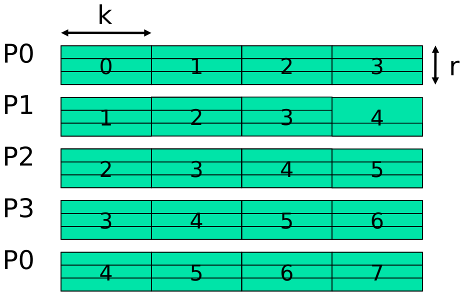


Reducing the Granularity More

- So far, we've allocated non-contiguous rows of the domain to each processor
- But we can reduce communication by allocating processors groups of contiguous rows
  - If two contiguous rows are on the same processors, there is no communication involved to update the cells of the second row
- Let us use say that we allocate blocks of rows of size r to each processor
  - We assume that r*p divides n
- Processor Pi holds rows j such that
  i = floor(j/r) mod p
- This is really a "block cyclic" allocation

Courtesy of Henri Casanova

Reducing the Granularity

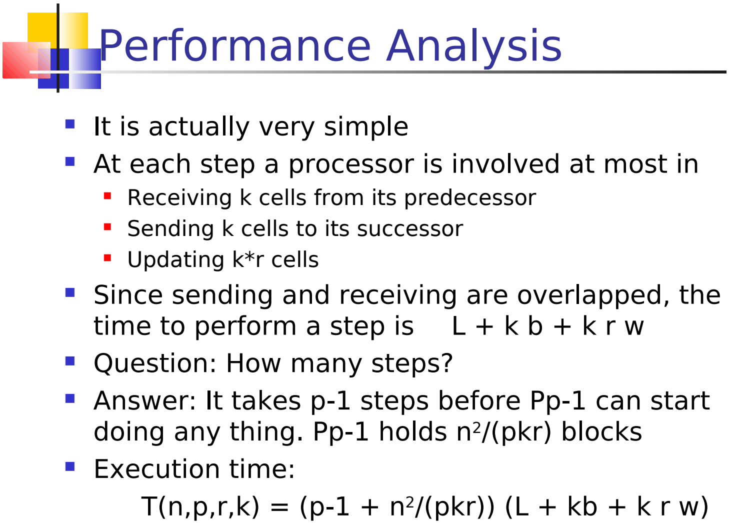


Courtesy of Henri Casanova

Idle Time?

- One question is: does any processor stay idle?
- Processor P_0 computes all values in its first block of rows in n/k algorithm steps
- After that, processor P_0 must wait for cell values from processor P_{p-1}
- But P_{p-1} cannot start computing before p steps
- Therefore:
  - If $p >= n/k$, P_0 is idle
  - If $p < n/k$, P_1 is not idle
- If $p < n/k$, then processors had better be able to buffer received cells while they are still computing
  - Possible increase in memory consumption

Courtesy of Henri Casanova



Performance Analysis

- It is actually very simple
- At each step a processor is involved at most in
  - Receiving k cells from its predecessor
  - Sending k cells to its successor
  - Updating k*r cells
- Since sending and receiving are overlapped, the time to perform a step is L + k b + k r w
- Question: How many steps?
- Answer: It takes p-1 steps before Pp-1 can start doing any thing. Pp-1 holds $n^2/(pkr)$ blocks
- Execution time:

$$T(n,p,r,k) = (p-1 + n^2/(pkr)) (L + kb + k r w)$$

Courtesy of Henri Casanova

Performance Analysis

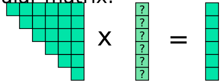
- Our naïve greedy algorithm had asymptotic efficiency equal to w / (w + L + b)
- This algorithm does better: Assympt. Eff = w / (w + L/rk + b/r)
  - Divide n^2w by p T(n,p,r,k)
  - And make n large
- In the formula for the efficiency we clearly see the effect of the granularity increase
- Asymptotic efficiency is higher
- But not equal to 1
- Therefore, this is a “difficult” application to parallelize
  - We can try to do the best we can by increasing r and k, but it’s never going to be perfect
- One can compute the optimal values of r and k using numerical solving
  - See the book for details

Courtesy of Henri Casanova

Outline

Solving Linear Systems of Eq.

- Method for solving Linear Systems
  - The need to solve linear systems arises in an estimated 75% of all scientific computing problems [Dahlquist 1974]
- Gaussian Elimination is perhaps the most well-known method
  - based on the fact that the solution of a linear system is invariant under scaling and under row additions
    - One can multiply a row of the matrix by a constant as long as one multiplies the corresponding element of the right-hand side by the same constant
    - One can add a row of the matrix to another one as long as one adds the corresponding elements of the right-hand side
  - Idea: scale and add equations so as to transform matrix A in an upper triangular matrix:



equation n-i has i unknowns, with

Courtesy of Henri Casanova

Gaussian Elimination

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 2 \\ 1 & 2 & -1 \end{bmatrix} x = \begin{bmatrix} 0 \\ 4 \\ 2 \end{bmatrix}$$

Subtract row 1 from rows 2 and 3

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 1 & -2 \end{bmatrix} x = \begin{bmatrix} 0 \\ 4 \\ 2 \end{bmatrix}$$

Multiple row 3 by 3 and add row 2

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 0 & -5 \end{bmatrix} x = \begin{bmatrix} 0 \\ 4 \\ 10 \end{bmatrix}$$

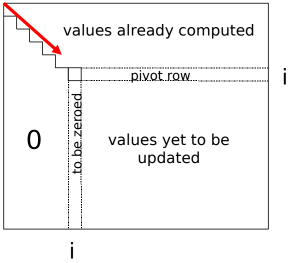
Solving equations in reverse order (backsolving)

$$\begin{aligned} -5x_3 &= 10 \\ -3x_2 + x_3 &= 4 \\ x_1 + x_2 + x_3 &= 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} x_3 &= -2 \\ x_2 &= -2 \\ x_1 &= 4 \end{aligned}$$

Courtesy of Henri Casanova

Gaussian Elimination

- The algorithm goes through the matrix from the top-left corner to the bottom-right corner
- the ith step eliminates non-zero sub-diagonal elements in column i, substracting the ith row scaled by a_{ji}/a_{ii} from row j, for j=i+1,..,n.



Courtesy of Henri Casanova

Sequential Gaussian Elimination

Simple sequential algorithm

```
// for each column i
// zero it out below the diagonal by adding
// multiples of row i to later rows
for i = 1 to n-1
    // for each row j below row i
    for j = i+1 to n
         // add a multiple of row i to row j
         for k = i to n
               A(j,k) = A(j,k) - (A(j,i)/A(i,i)) * A(i,k)
```

- Several “tricks” that do not change the spirit of the algorithm but make implementation easier and/or more efficient
  - Right-hand side is typically kept in column n+1 of the matrix and one speaks of an *augmented matrix*
  - Compute the A(i,j)/A(i,i) term outside of the loop

Courtesy of Henri Casanova

Pivoting: Motivation

- A few pathological cases

0	1
1	1

- Division by small numbers $\rightarrow$ round-off error in computer arithmetic
- Consider the following system
  $0.0001x_1 + x_2 = 1.000$
  $x_1 \quad + x_2 = 2.000$
- exact solution: $x_1=1.00010$ and $x2 = 0.99990$
- say we round off after 3 digits after the decimal point
- Multiply the first equation by 10^4 and subtract it from the second equation
- $(1 - 1)x_1 + (1 - 10^4)x_2 = 2 - 10^4$
- But, in finite precision with only 3 digits:
  - $1 - 10^4 = -0.9999 \text{ E+4} \sim -0.999 \text{ E+4}$
  - $2 - 10^4 = -0.9998 \text{ E+4} \sim -0.999 \text{ E+4}$
- Therefore, $x_2 = 1$ and $x_1 = 0$ (from the first equation)

Courtesy of Henri Casanova

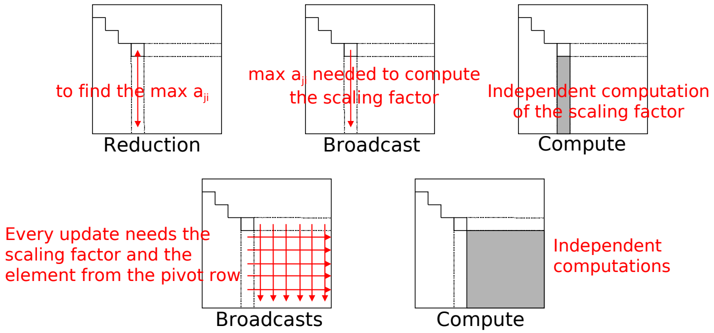
Partial Pivoting

- One can just swap rows
  $x_1 + x_2 = 2.000$
  $0.0001x_1 + x_2 = 1.000$
- Multiple the first equation my 0.0001 and subtract it from the second equation gives:
  $(1 - 0.0001)x2 = 1 - 0.0001$
  $0.9999\, x_2 = 0.9999 \Rightarrow x_2 = 1$
  and then $x_1 = 1$
- Final solution is closer to the real solution. (Magical?)
- Partial Pivoting
  - For *numerical stability*, one doesn't go in order, but pick the next row in rows i to n that has the largest element in row i
  - This row is swapped with row i (along with elements of the right hand side) before the subtractions
    - the swap is not done in memory but rather one keeps an indirection array
- Total Pivoting
  - Look for the greatest element ANYWHERE in the matrix
  - Swap columns
  - Swap rows
- Numerical stability is really a difficult field

Courtesy of Henri Casanova

Parallel Gaussian Elimination?

- Assume that we have one processor per matrix element

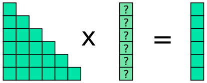

to find the max a_{ji}
Reduction
max a_{ji} needed to compute the scaling factor
Broadcast
Independent computation of the scaling factor
Compute
Every update needs the scaling factor and the element from the pivot row
Broadcasts
Independent computations
Compute

Courtesy of Henri Casanova

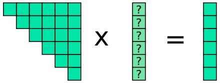
LU Factorization (Section 4.4)

- Gaussian Elimination is simple but
  - What if we have to solve many Ax = b systems for different values of b?
    - This happens a LOT in real applications
- Another method is the "LU Factorization"
- Ax = b
- Say we could rewrite A = L U, where L is a lower triangular matrix, and U is an upper triangular matrix $O(n^3)$
- Then Ax = b is written L U x = b
- Solve L y = b $O(n^2)$
- Solve U x = y $O(n^2)$

triangular system solves are easy



equation i has i unknowns

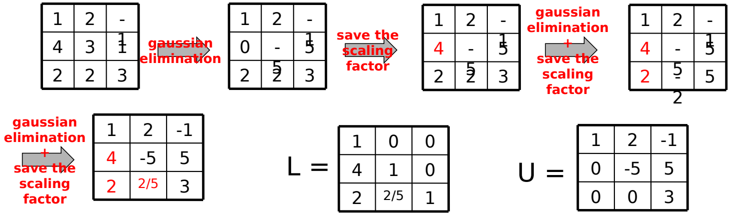


equation n-i has i unknowns

Courtesy of Henri Casanova

LU Factorization: Principle

- It works just like the Gaussian Elimination, but instead of zeroing out elements, one "saves" scaling coefficients.


gaussian elimination
save the scaling factor
gaussian elimination + save the scaling factor
gaussian elimination + save the scaling factor

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 2 & 2/5 & 1 \end{pmatrix} \qquad U = \begin{pmatrix} 1 & 2 & -1 \\ 0 & -5 & 5 \\ 0 & 0 & 3 \end{pmatrix}$$

- Magically, $A = L \times U$!
- Should be done with pivoting as well

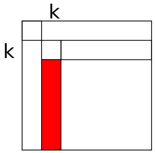
Courtesy of Henri Casanova

LU Factorization

- We're going to look at the simplest possible version
  - No pivoting:just creates a bunch of indirections that are easy but make the code look complicated without changing the overall principle

```
LU-sequential(A,n) {
  for k = 0 to n-2 {
     // preparing column k
     for i = k+1 to n-1
        $a_{ik} \leftarrow -a_{ik} / a_{kk}$
     for j = k+1 to n-1
        // Task $T_{kj}$: update of column j
        for i=k+1 to n-1
           $a_{ij} \leftarrow a_{ij} + a_{ik} * a_{kj}$
  }
}
```

stores the scaling factors



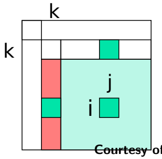
Courtesy of Henri Casanova

LU Factorization

- We're going to look at the simplest possible version
  - No pivoting:just creates a bunch of indirections that are easy but make the code look complicated without changing the overall principle

```
LU-sequential(A,n) {
  for k = 0 to n-2 {
     // preparing column k
     for i = k+1 to n-1
        $a_{ik} \leftarrow -a_{ik} / a_{kk}$
     for j = k+1 to n-1
        // Task $T_{kj}$: update of column j
        for i=k+1 to n-1
           $a_{ij} \leftarrow a_{ij} + a_{ik} * a_{kj}$
  }
}
```

update



Courtesy of Henri Casanova

Parallel LU on a ring

- Since the algorithm operates by columns from left to right, we should distribute columns to processors
- Principle of the algorithm
  - At each step, the processor that owns column k does the "prepare" task and then broadcasts the bottom part of column k to all others
    - Annoying if the matrix is stored in row-major fashion
    - Remember that one is free to store the matrix in anyway one wants, as long as it's coherent and that the right output is generated
  - After the broadcast, the other processors can then update their data.
- Assume there is a function alloc(k) that returns the rank of the processor that owns column k
  - Basically so that we don't clutter our program with too many global-to-local index translations
- In fact, we will first write everything in terms of global indices, as to avoid all annoying index arithmetic

LU-broadcast algorithm

```
LU-broadcast(A,n) {
  q ← MY_NUM()
  p ← NUM_PROCS()
  for k = 0 to n-2 {
    if (alloc(k) == q)
       // preparing column k
       for i = k+1 to n-1
          buffer[i-k-1] ← $a_{ik}$ ← -$a_{ik}$ / $a_{kk}$
    broadcast(alloc(k),buffer,n-k-1)
    for j = k+1 to n-1
      if (alloc(j) == q)
        // update of column j
        for i=k+1 to n-1
          $a_{ij}$ ← $a_{ij}$ + buffer[i-k-1] * $a_{kj}$
  }
}
```

Courtesy of Henri Casanova

Dealing with local indices

- Assume that p divides n
- Each processor needs to store r=n/p columns and its local indices go from 0 to r-1
- After step k, only columns with indices greater than k will be used
- Simple idea: use a local index, l, that everyone initializes to 0
- At step k, processor alloc(k) increases its local index so that next time it will point to its next local column

Courtesy of Henri Casanova

LU-broadcast algorithm

```
...
  double a[n-1][r-1];

  q ← MY_NUM()
  p ← NUM_PROCS()
  l ← 0
  for k = 0 to n-2 {
    if (alloc(k) == q)
        for i = k+1 to n-1
          buffer[i-k-1] ← a[i,k] ← -a[i,l] / a[k,l]
        l ← l+1
    broadcast(alloc(k),buffer,n-k-1)
    for j = l to r-1
        for i=k+1 to n-1
          a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
  }
}
```

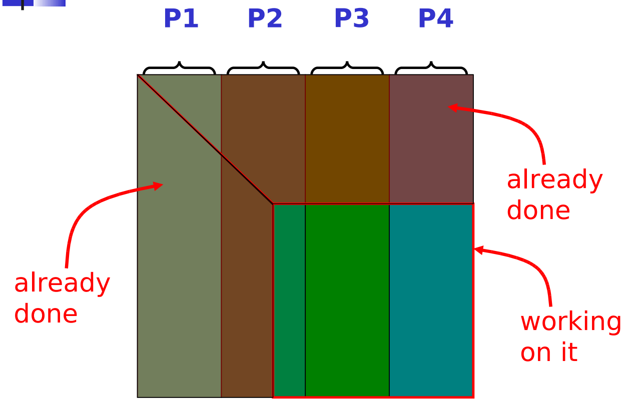
Courtesy of Henri Casanova

What about the Alloc function?

- One thing we have left completely unspecified is how to write the alloc function: how are columns distributed among processors
- There are two complications:
  - The amount of data to process varies throughout the algorithm's execution
    - At step k, columns k+1 to n-1 are updated
    - Fewer and fewer columns to update
  - The amount of computation varies among columns
    - e.g., column n-1 is updated more often than column 2
    - Holding columns on the right of the matrix leads to much more work
- There is a strong need for load balancing
  - All processes should do the same amount of work

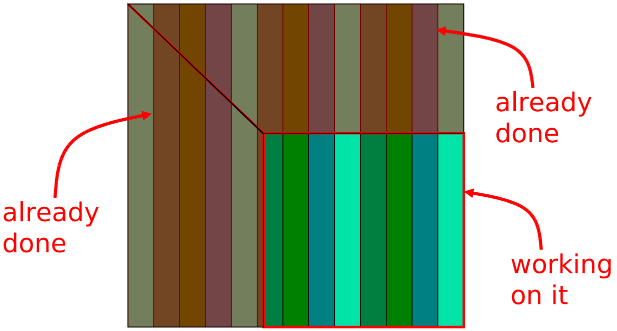
Courtesy of Henri Casanova

Bad load balancing



Courtesy of Henri Casanova

Good Load Balancing?



Cyclic distribution

Courtesy of Henri Casanova

Proof that load balancing is good

- The computation consists of two types of operations
  - column preparations
  - matrix element updates
- There are many more updates than preparations, so we really care about good balancing of the preparations
- Consider column j
- Let's count the number of updates performed by the processor holding column j
- Column j is updated at steps k=0, ..., j-1
- At step k, elements i=k+1, ..., n-1 are updates
  - indices start at 0
- Therefore, at step k, the update of column j entails n-k-1 updates
- The total number of updates for column j in the execution is:

$$\sum_{k=0}^{j-1}(n-k-1) = j(n-1) - \frac{j(j-1)}{2}$$

Courtesy of Henri Casanova

Proof that load balancing is good

- Consider processor P_i, which holds columns lp+i for l=0, ... , n/p -1
- Processor P_i needs to perform this many updates:

$$\sum_{l=0}^{n/p-1} \left((lp+i)(n-1) - \frac{(lp+i)(lp+i-1)}{2}\right)$$

- Turns out this can be computed
  - separate terms
  - use formulas for sums of integers and sums of squares
- What it all boils down to is:

$$\frac{n^3}{3p} + O(n^2)$$

- This does not depend on i !!
- Therefore it is (asymptotically) the same for all P_i processors
- Therefore we have (asymptotically) perfect load balancing!

Courtesy of Henri Casanova

Load-balanced program

```
...
  double a[n-1][r-1];

  q ← MY_NUM()
  p ← NUM_PROCS()
  l ← 0
  for k = 0 to n-2 {
    if (k mod p == q)
        for i = k+1 to n-1
          buffer[i-k-1] ← a[i,k] ← -a[i,l] / a[k,l]
        l ← l+1
    broadcast(alloc(k),buffer,n-k-1)
    for j = l to r-1
        for i=k+1 to n-1
          a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
  }
}
```

Courtesy of Henri Casanova

Performance Analysis

- How long does this code take to run?
- This is not an easy question because there are many tasks and many communications
- A little bit of analysis shows that the execution time is the sum of three terms
  - n-1 communications: $n L + (n^2/2) b + O(1)$
  - n-1 column preparations: $(n^2/2) w' + O(1)$
  - column updates: $(n^3/3p) w + O(n^2)$
- Therefore, the execution time is $\sim (n^3/3p)\, w$
- Note that the sequential time is: $(n^3/3)\, w$
- Therefore, we have perfect asymptotic efficiency!
- This is good, but isn't always the best in practice
- How can we improve this algorithm?

Courtesy of Henri Casanova

Pipelining on the Ring

- So far, the algorithm we've used a simple broadcast
- Nothing was specific to being on a ring of processors and it's portable
  - in fact you could just write raw MPI that just looks like our pseudo-code and have a very limited, inefficient for small n, LU factorization that works only for some number of processors
- But it's not efficient
  - The n-1 communication steps are not overlapped with computations
  - Therefore Amdahl's law, etc.
- Turns out that on a ring, with a cyclic distribution of the columns, one can interleave pieces of the broadcast with the computation
  - It almost looks like inserting the source code from the

Courtesy of Henri Casanova

Part V

A Complete Example on an Heterogeneous Ring

The Context: Distributed Heterogeneous Platforms

How to embed a ring in a complex network [LRRV04].
Sources of problems

- Heterogeneity of processors (computational power, memory, etc.)
- Heterogeneity of communications links.
- Irregularity of interconnection network.

Targeted Applications: Iterative Algorithms

- A set of data (typically, a matrix)
- Structure of the algorithms:
  1. Each processor performs a computation on its chunk of data
  2. Each processor exchange the "border" of its chunk of data with its neighbor processors
  3. We go back at Step 1

Targeted Applications: Iterative Algorithms

- A set of data (typically, a matrix)
- Structure of the algorithms:
  1. Each processor performs a computation on its chunk of data
  2. Each processor exchange the “border” of its chunk of data with its neighbor processors
  3. We go back at Step 1

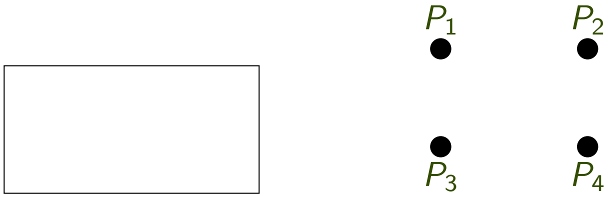
Question: how can we efficiently execute such an algorithm on such a platform?

The Questions

- Which processors should be used ?
- What amount of data should we give them ?
- How do we cut the set of data ?

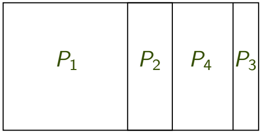
First of All, a Simplification: Slicing the Data

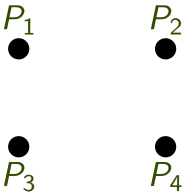
- Data: a 2-D array



First of All, a Simplification: Slicing the Data

- Data: a 2-D array

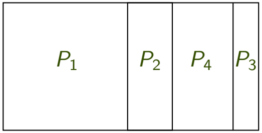


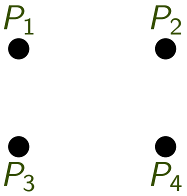


- Unidimensional cutting into vertical slices

First of All, a Simplification: Slicing the Data

- Data: a 2-D array

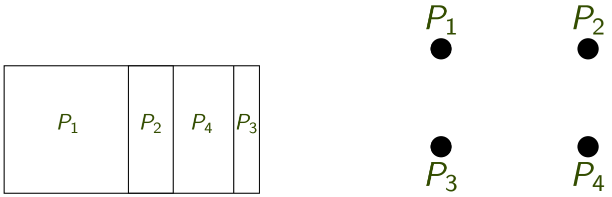




- Unidimensional cutting into vertical slices
- Consequences:

First of All, a Simplification: Slicing the Data

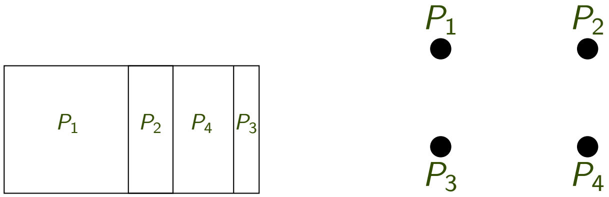
- Data: a 2-D array



- Unidimensional cutting into vertical slices
- Consequences:
  1. Borders and neighbors are easily defined

First of All, a Simplification: Slicing the Data

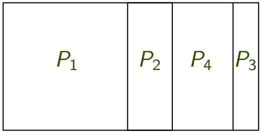
- Data: a 2-D array

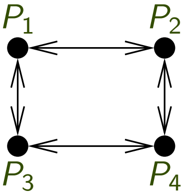


- Unidimensional cutting into vertical slices
- Consequences:
  1. Borders and neighbors are easily defined
  2. Constant volume of data exchanged between neighbors: D_c

First of All, a Simplification: Slicing the Data

- Data: a 2-D array





- Unidimensional cutting into vertical slices
- Consequences:
  1. Borders and neighbors are easily defined
  2. Constant volume of data exchanged between neighbors: D_c
  3. Processors are virtually organized into a ring

Notations

- Processors: P_1, ..., P_p
- Processor P_i executes a unit task in a time w_i
- Overall amount of work D_w;
  Share of P_i: $\alpha_i.D_w$ processed in a time $\alpha_i.D_w.w_i$
  ($\alpha_i \geqslant 0$, $\sum_j \alpha_j = 1$)
- Cost of a unit-size communication from P_i to P_j: $c_{i,j}$
- Cost of a sending from P_i to its successor in the ring: $D_c.c_{i,\text{succ}(i)}$

Communications: 1-Port Model (Full-Duplex)

A processor can:

- send at most one message at any time;
- receive at most one message at any time;
- send and receive a message simultaneously.

Objective

1. Select q processors among p

Objective

1. Select q processors among p
2. Order them into a ring

Objective

1. Select q processors among p
2. Order them into a ring
3. Distribute the data among them

Objective

1. Select q processors among p
2. Order them into a ring
3. Distribute the data among them

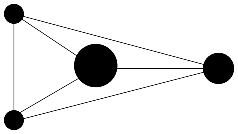
So as to minimize:

$$\max_{1 \leqslant i \leqslant p} \mathbb{I}\{i\}[\alpha_i.D_w.w_i + D_c.(c_{i,\text{pred}(i)} + c_{i,\text{succ}(i)})]$$

Where $\mathbb{I}\{i\}[x] = x$ if P_i participates in the computation, and 0 otherwise

Special Hypotheses

1. There exists a communication link between any two processors
2. All links have the same capacity
   ($\exists c, \forall i, j \ c_{i,j} = c$)



Consequences

- Either the most powerful processor performs all the work, or all the processors participate

Consequences

- Either the most powerful processor performs all the work, or all the processors participate
- If all processors participate, all end their share of work simultaneously

Consequences

- Either the most powerful processor performs all the work, or all the processors participate
- If all processors participate, all end their share of work simultaneously($\exists\tau, \quad \alpha_i.D_w.w_i = \tau$, so $1 = \sum_i \frac{\tau}{D_w.w_i}$)

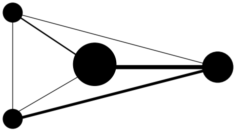
Consequences

- Either the most powerful processor performs all the work, or all the processors participate
- If all processors participate, all end their share of work simultaneously($\exists\tau, \quad \alpha_i.D_w.w_i = \tau$, so $1 = \sum_i \frac{\tau}{D_w.w_i}$)
- Time of the optimal solution:

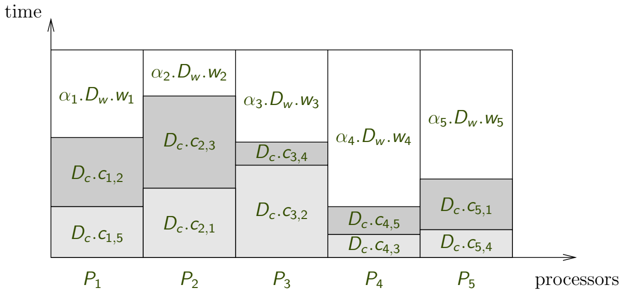
$$T_{\text{step}} = \min\left\{D_w.w_{\min}, D_w.\frac{1}{\sum_i \frac{1}{w_i}} + 2.D_c.c\right\}$$

Special hypothesis

1. There exists a communication link between any two processors



All the Processors Participate: Study (1)



All processors end simultaneously

All the Processors Participate: Study (2)

- All processors end simultaneously

$$T_{\text{step}} = \alpha_i . D_w . w_i + D_c . (c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)})$$

All the Processors Participate: Study (2)

- All processors end simultaneously

$$T_{\text{step}} = \alpha_i . D_w . w_i + D_c . (c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)})$$

- $\sum_{i=1}^{p} \alpha_i = 1 \;\Rightarrow\; \sum_{i=1}^{p} \frac{T_{\text{step}} - D_c . (c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)})}{D_w . w_i} = 1.$ Thus

$$\frac{T_{\text{step}}}{D_w . w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$$

where $w_{\text{cumul}} = \frac{1}{\sum_i \frac{1}{w_i}}$

All the Processors Participate: Interpretation

$$\frac{T_{\text{step}}}{D_w . w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$$

All the Processors Participate: Interpretation

$$\frac{T_{\text{step}}}{D_w . w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$$

T_{step} is minimal when $\sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$ is minimal

All the Processors Participate: Interpretation

$$\frac{T_{\text{step}}}{D_w.w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$$

T_{step} is minimal when $\displaystyle\sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$ is minimal

Look for an hamiltonian cycle of minimal weight in a graph where the edge from P_i to P_j has a weight of $d_{i,j} = \frac{c_{i,j}}{w_i} + \frac{c_{j,i}}{w_j}$

All the Processors Participate: Interpretation

$$\frac{T_{\text{step}}}{D_w . w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$$

T_{step} is minimal when $\displaystyle\sum_{i=1}^{p} \frac{c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)}}{w_i}$ is minimal

Look for an hamiltonian cycle of minimal weight in a graph where the edge from P_i to P_j has a weight of $d_{i,j} = \frac{c_{i,j}}{w_i} + \frac{c_{j,i}}{w_j}$

NP-complete problem

All the Processors Participate: Linear Program

$$\textsc{Minimize}\ \sum_{i=1}^{p}\sum_{j=1}^{p} d_{i,j}.x_{i,j},$$

$$\textsc{satisfying the (in)equations}$$

$$\left\{\begin{array}{lll}(1) & \sum_{j=1}^{p} x_{i,j} = 1 & 1 \leqslant i \leqslant p \\ (2) & \sum_{i=1}^{p} x_{i,j} = 1 & 1 \leqslant j \leqslant p \\ (3) & x_{i,j} \in \{0,1\} & 1 \leqslant i,j \leqslant p \\ (4) & u_i - u_j + p.x_{i,j} \leqslant p-1 & 2 \leqslant i,j \leqslant p, i \neq j \\ (5) & u_i \text{ integer}, u_i \geqslant 0 & 2 \leqslant i \leqslant p\end{array}\right.$$

$x_{i,j} = 1$ if, and only if, the edge from P_i to P_j is used

General Case: Linear program

Best ring made of q processors

MINIMIZE T SATISFYING THE (IN)EQUATIONS

$$\left\{\begin{array}{ll}
(1)\ x_{i,j} \in \{0,1\} & 1 \leqslant i,j \leqslant p \\
(2)\ \sum_{i=1}^{p} x_{i,j} \leqslant 1 & 1 \leqslant j \leqslant p \\
(3)\ \sum_{i=1}^{p}\sum_{j=1}^{p} x_{i,j} = q & \\
(4)\ \sum_{i=1}^{p} x_{i,j} = \sum_{i=1}^{p} x_{j,i} & 1 \leqslant j \leqslant p \\
 & \\
(5)\ \sum_{i=1}^{p} \alpha_i = 1 & \\
(6)\ \alpha_i \leqslant \sum_{j=1}^{p} x_{i,j} & 1 \leqslant i \leqslant p \\
(7)\ \alpha_i.w_i + \frac{D_c}{D_w}\sum_{j=1}^{p}(x_{i,j}c_{i,j} + x_{j,i}c_{j,i}) \leqslant T & 1 \leqslant i \leqslant p \\
 & \\
(8)\ \sum_{i=1}^{p} y_i = 1 & \\
(9)\ -p.y_i - p.y_j + u_i - u_j + q.x_{i,j} \leqslant q-1 & 1 \leqslant i,j \leqslant p, i \neq j \\
(10)\ y_i \in \{0,1\} & 1 \leqslant i \leqslant p \\
(11)\ u_i \text{ integer}, u_i \geqslant 0 & 1 \leqslant i \leqslant p
\end{array}\right.$$

Linear Programming

- Problems with rational variables: can be solved in polynomial time (in the size of the problem).
- Problems with integer variables: solved in exponential time in the worst case.
- No relaxation in rationals seems possible here...

And, in Practice ?

All processors participate. One can use a heuristic to solve the traveling salesman problem (as Lin-Kernighan's one)

And, in Practice ?

All processors participate. One can use a heuristic to solve the traveling salesman problem (as Lin-Kernighan's one)
No guarantee, but excellent results in practice.

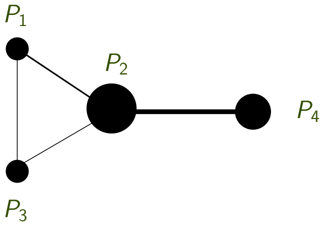
And, in Practice ?

All processors participate. One can use a heuristic to solve the traveling salesman problem (as Lin-Kernighan's one)
No guarantee, but excellent results in practice.

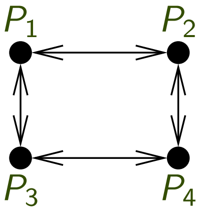
General case.

1. Exhaustive search: feasible until a dozen of processors...
2. Greedy heuristic: initially we take the best pair of processors; for a given ring we try to insert any unused processor in between any pair of neighbor processors in the ring...

New Difficulty: Communication Links Sharing

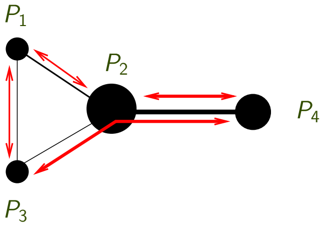


Heterogeneous platform

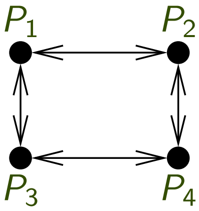


Virtual ring

New Difficulty: Communication Links Sharing

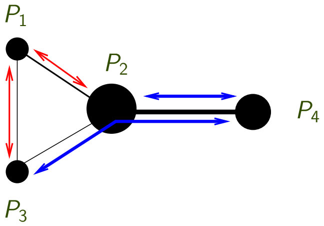


Heterogeneous platform

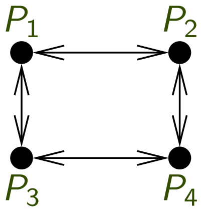


Virtual ring

New Difficulty: Communication Links Sharing

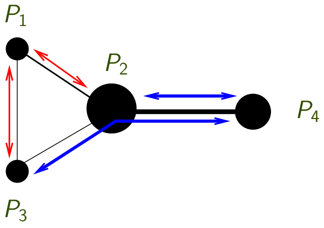


Heterogeneous platform

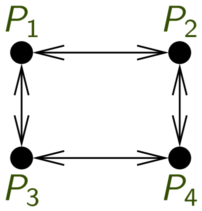


Virtual ring

New Difficulty: Communication Links Sharing



Heterogeneous platform



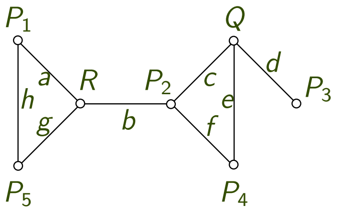
Virtual ring

We must take communication link sharing into account.

New Notations

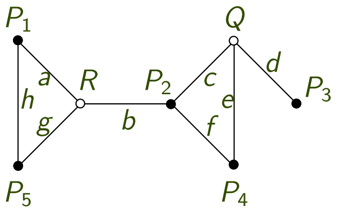
- A set of communications links: e_1, ..., e_n
- Bandwidth of link e_m: b_{e_m}
- There is a path $\mathcal{S}_i$ from P_i to $P_{\text{succ}(i)}$ in the network
  - $\mathcal{S}_i$ uses a fraction $s_{i,m}$ of the bandwidth b_{e_m} of link e_m
  - P_i needs a time $D_c.\dfrac{1}{\min_{e_m \in \mathcal{S}_i} s_{i,m}}$ to send to its successor a message of size D_c
  - Constraints on the bandwidth of e_m: $\displaystyle\sum_{1 \leqslant i \leqslant p} s_{i,m} \leqslant b_{e_m}$
- Symmetrically, there is a path $\mathcal{P}_i$ from P_i to $P_{\text{pred}(i)}$ in the network, which uses a fraction $p_{i,m}$ of the bandwidth b_{e_m} of link e_m

Toy Example: Choosing the Ring



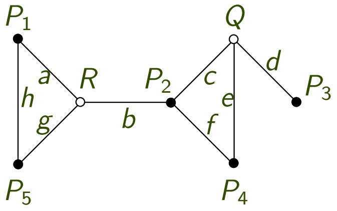
- 7 processors and 8 bidirectional communications links
- We choose a ring of 5 processors:
  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$ (we use neither Q, nor R)

Toy Example: Choosing the Ring

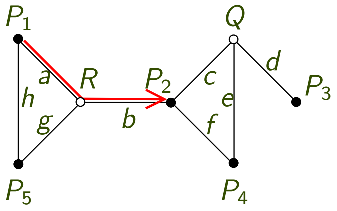


- 7 processors and 8 bidirectional communications links
- We choose a ring of 5 processors:
  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$ (we use neither Q, nor R)

Toy Example: Choosing the Paths

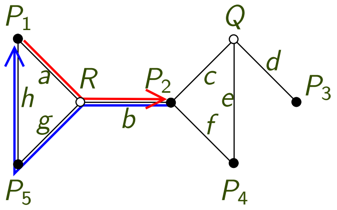


Toy Example: Choosing the Paths



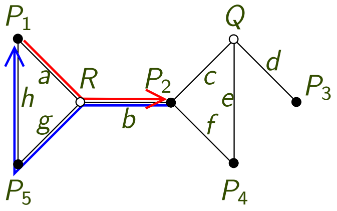
From P_1 to P_2, we use the links a and b: $\mathcal{S}_1 = \{a, b\}$.

Toy Example: Choosing the Paths



From P_1 to P_2, we use the links a and b: $\mathcal{S}_1 = \{a, b\}$.
From P_2 to P_1, we use the links b, g and h: $\mathcal{P}_2 = \{b, g, h\}$.

Toy Example: Choosing the Paths



From P_1 to P_2, we use the links a and b: $\mathcal{S}_1 = \{a, b\}$.
From P_2 to P_1, we use the links b, g and h: $\mathcal{P}_2 = \{b, g, h\}$.

From P_1: to P_2, $\mathcal{S}_1 = \{a, b\}$ and to P_5, $\mathcal{P}_1 = \{h\}$
From P_2: to P_3, $\mathcal{S}_2 = \{c, d\}$ and to P_1, $\mathcal{P}_2 = \{b, g, h\}$
From P_3: to P_4, $\mathcal{S}_3 = \{d, e\}$ and to P_2, $\mathcal{P}_3 = \{d, e, f\}$
From P_4: to P_5, $\mathcal{S}_4 = \{f, b, g\}$ and to P_3, $\mathcal{P}_4 = \{e, d\}$
From P_5: to P_1, $\mathcal{S}_5 = \{h\}$ and to P_4, $\mathcal{P}_5 = \{g, b, f\}$

Toy Example: Bandwidth Sharing

From P_1 to P_2 we use links a and b: $c_{1,2} = \frac{1}{\min(s_{1,a}, s_{1,b})}$.
From P_1 to P_5 we use the link h: $c_{1,5} = \frac{1}{p_{1,h}}$.

Toy Example: Bandwidth Sharing

From P_1 to P_2 we use links a and b: $c_{1,2} = \frac{1}{\min(s_{1,a}, s_{1,b})}$.
From P_1 to P_5 we use the link h: $c_{1,5} = \frac{1}{p_{1,h}}$.

Set of all sharing constraints:

Lien a: $s_{1,a} \leqslant b_a$
Lien b: $s_{1,b} + s_{4,b} + p_{2,b} + p_{5,b} \leqslant b_b$
Lien c: $s_{2,c} \leqslant b_c$
Lien d: $s_{2,d} + s_{3,d} + p_{3,d} + p_{4,d} \leqslant b_d$
Lien e: $s_{3,e} + p_{3,e} + p_{4,e} \leqslant b_e$
Lien f: $s_{4,f} + p_{3,f} + p_{5,f} \leqslant b_f$
Lien g: $s_{4,g} + p_{2,g} + p_{5,g} \leqslant b_g$
Lien h: $s_{5,h} + p_{1,h} + p_{2,h} \leqslant b_h$

Toy Example: Final Quadratic System

MINIMIZE $\max_{1\leqslant i\leqslant 5} \left(\alpha_i.D_w.w_i + D_c.(c_{i,i-1}+c_{i,i+1})\right)$ UNDER THE CONSTRAINTS

$$\left\{\begin{array}{lll}
\sum_{i=1}^{5}\alpha_i = 1 & & \\
s_{1,a}\leqslant b_a & s_{1,b}+s_{4,b}+p_{2,b}+p_{5,b}\leqslant b_b & s_{2,c}\leqslant b_c \\
s_{2,d}+s_{3,d}+p_{3,d}+p_{4,d}\leqslant b_d & s_{3,e}+p_{3,e}+p_{4,e}\leqslant b_e & s_{4,f}+p_{3,f}+p_{5,f}\leqslant b_f \\
s_{4,g}+p_{2,g}+p_{5,g}\leqslant b_g & s_{5,h}+p_{1,h}+p_{2,h}\leqslant b_h & \\
s_{1,a}.c_{1,2}\geqslant 1 & s_{1,b}.c_{1,2}\geqslant 1 & p_{1,h}.c_{1,5}\geqslant 1 \\
s_{2,c}.c_{2,3}\geqslant 1 & s_{2,d}.c_{2,3}\geqslant 1 & p_{2,b}.c_{2,1}\geqslant 1 \\
p_{2,g}.c_{2,1}\geqslant 1 & p_{2,h}.c_{2,1}\geqslant 1 & s_{3,d}.c_{3,4}\geqslant 1 \\
s_{3,e}.c_{3,4}\geqslant 1 & p_{3,d}.c_{3,2}\geqslant 1 & p_{3,e}.c_{3,2}\geqslant 1 \\
p_{3,f}.c_{3,2}\geqslant 1 & s_{4,f}.c_{4,5}\geqslant 1 & s_{4,b}.c_{4,5}\geqslant 1 \\
s_{4,g}.c_{4,5}\geqslant 1 & p_{4,e}.c_{4,3}\geqslant 1 & p_{4,d}.c_{4,3}\geqslant 1 \\
s_{5,h}.c_{5,1}\geqslant 1 & p_{5,g}.c_{5,4}\geqslant 1 & p_{5,b}.c_{5,4}\geqslant 1 \\
p_{5,f}.c_{5,4}\geqslant 1 & &
\end{array}\right.$$

Toy Example: Conclusion

The problem sums up to a quadratic system if

1. The processors are selected;
2. The processors are ordered into a ring;
3. The communication paths between the processors are known.

In other words: a quadratic system if the ring is known.

Toy Example: Conclusion

The problem sums up to a quadratic system if

1. The processors are selected;
2. The processors are ordered into a ring;
3. The communication paths between the processors are known.

In other words: a quadratic system if the ring is known.

If the ring is known:

- Complete graph: closed-form expression;
- General graph: quadratic system.

And, in Practice ?

We adapt our greedy heuristic:

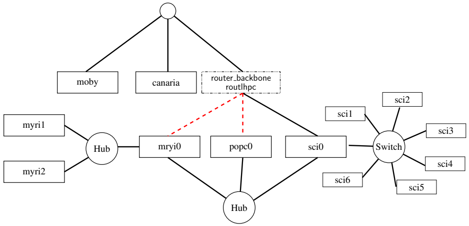
1. Initially: best pair of processors
2. For each processor P_k (not already included in the ring)
   - For each pair (P_i, P_j) of neighbors in the ring
     1. We build the graph of the unused bandwidths (Without considering the paths between P_i and P_j)
     2. We compute the shortest paths (in terms of bandwidth) between P_k and P_i and P_j
     3. We evaluate the solution
3. We keep the best solution found at step 2 and we start again

$+$ refinements (*max-min fairness*, quadratic solving).

Is This Meaningful ?

- No guarantee, neither theoretical, nor practical
- Simple solution:
  1. we build the complete graph whose edges are labeled with the bandwidths of the best communication paths
  2. we apply the heuristic for complete graphs
  3. we allocate the bandwidths

Example: an Actual Platform (Lyon)



Topology

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
0.0206	0.0206	0.0206	0.0206	0.0291	0.0206	0.0087	0.0206	0.0206

P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
0.0206	0.0206	0.0206	0.0291	0.0451	0	0	0

Processors processing times (in seconds par megaflop)

Results

First heuristic building the ring without taking link sharing into account

Second heuristic taking into account link sharing (and with quadratic programing)

Ratio D_c/D_w	H1	H2	Gain
0.64	0.008738 (1)	0.008738 (1)	0%
0.064	0.018837 (13)	0.006639 (14)	64.75%
0.0064	0.003819 (13)	0.001975 (14)	48.28%

Ratio D_c/D_w	H1	H2	Gain
0.64	0.005825 (1)	0.005825 (1)	0 %
0.064	0.027919 (8)	0.004865 (6)	82.57%
0.0064	0.007218 (13)	0.001608 (8)	77.72%

Table: T_{step}/D_w for each heuristic on Lyon's and Strasbourg's platforms (the numbers in parentheses show the size of the rings built).

Conclusion

Even though this is a very basic application, it illustrates many difficulties encountered when:

- Processors have different characteristics
- Communications links have different characteristics
- There is an irregular interconnection network with complex bandwidth sharing issues.

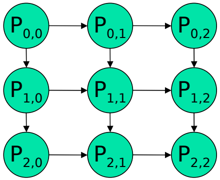
We need to use a realistic model of networks... Even though a more realistic model leads to a much more complicated problem, this is worth the effort as derived solutions are more efficient in practice.

Part VI

Algorithms on a Grid

Outline

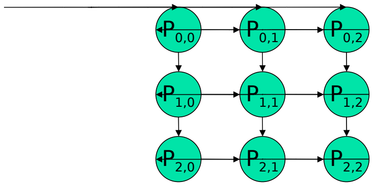
2-D Grid (Chapter 5)


$$\begin{tikzcd}
P_{0,0} \arrow[r] \arrow[d] & P_{0,1} \arrow[r] \arrow[d] & P_{0,2} \arrow[d] \\
P_{1,0} \arrow[r] \arrow[d] & P_{1,1} \arrow[r] \arrow[d] & P_{1,2} \arrow[d] \\
P_{2,0} \arrow[r] & P_{2,1} \arrow[r] & P_{2,2}
\end{tikzcd}$$

- Consider $p=q^2$ processors
- We can think of them arranged in a square grid
  - A rectangular grid is also possible, but we'll stick to square grids for most of our algorithms
- Each processor is identified as $P_{i,j}$
  - i: processor row
  - J: processor column

Courtesy of Henri Casanova

2-D Torus



- Wrap-around links from edge to edge
- Each processor belongs to 2 different rings
  - Will make it possible to reuse algorithms we develop for the ring topology
- Mono-directional links OR Bi-directional links
  - Depending on what we need the algorithm to do and on what makes sense for the physical platform

Courtesy of Henri Casanova

Concurrency of Comm. and Comp.

- When developing performance models we will assume that a processor can do all three activities in parallel
  - Compute
  - Send
  - Receive
- What about the bi-directional assumption?
  - Two models
    - Half-duplex: two messages on the same link going in opposite directions contend for the link's bandwidth
    - Full-duplex: it's as if we had two links in between each neighbor processors
  - The validity of the assumption depends on the platform

Courtesy of Henri Casanova

Multiple concurrent communications?

- Now that we have 4 (logical) links at each processor, we need to decide how many concurrent communications can happen at the same time
  - There could be 4 sends and 4 receives in the bi-directional link model
- If we assume that 4 sends and 4 receives can happened concurrently without loss of performance, we have a *multi-port* model
- If we only allow 1 send and 1 receive to occur concurrently we have a *single-port* model

Courtesy of Henri Casanova

So what?

- We have many options:
  - Grid or torus
  - Mono- or bi-directional
  - Single-or multi-port
  - Half- or full-duplex
- We'll mostly use the torus, bi-directional, full-duplex assumption
- We'll discuss the multi-port and the single-port assumptions
- As usual, it's straightforward to modify the performance analyses to match with whichever assumption makes sense for the physical platform

Courtesy of Henri Casanova

How realistic is a grid topology?

- Some parallel computers are built as physical grids (2-D or 3-D)
  - Example: IBM’s Blue Gene/L
- If the platform uses a switch with all-to-all communication links, then a grid is actually not a bad assumption
  - Although the full-duplex or multi-port assumptions may not hold
- We will see that even if the physical platform is a shared single medium (e.g., a non-switched Ethernet), it’s sometimes preferable to think of it as a grid when developing algorithms!

Courtesy of Henri Casanova

Communication on a Grid

- As usual we won’t write MPI here, but some pseudo code
- A processor can call two functions to known where it is in the grid:
  - My_Proc_Row()
  - My_Proc_Col()
- A processor can find out how many processors there are in total by:
  - Num_Procs()
  - Recall that here we assume we have a square grid
  - In programming assignment we may need to use a rectangular grid

Courtesy of Henri Casanova

Communication on the Grid

- We have two point-to-point functions
  - Send(dest, addr, L)
  - Recv(src, addr, L)
- We will see that it's convenient to have broadcast algorithms within processor rows or processor columns
  - BroadcastRow(i, j, srcaddr, dstaddr, L)
  - BroadcastCol(i, j, srcaddr, dstaddr, L)
- We assume that a a call to these functions by a processor not on the relevant processor row or column simply returns immediately
- How do we implement these broadcasts?

Courtesy of Henri Casanova

Row and Col Broadcasts?

- If we have a torus
  - If we have mono-directional links, then we can reuse the broadcast that we developed on a ring of processors
    - Either pipelined or not
  - It we have bi-directional links AND a multi-port model, we can improved performance by going both-ways simultaneously on the ring
    - We'll see that the asymptotic performance is not changed
- If we have a grid
  - If links are bi-directional then messages can be sent both ways from the source processor
    - Either concurrently or not depending on whether we have a one-port or multi-port model
  - If links are mono-directional, then we can't implement the broadcasts at all

Courtesy of Henri Casanova

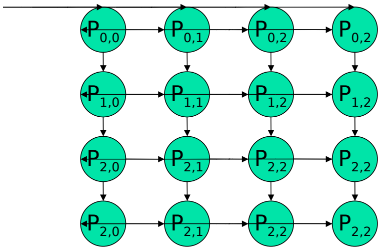
Outline

Matrix Multiplication on a Grid

- Matrix multiplication on a Grid has been studied a lot because
  - Multiplying huge matrices fast is always important in many, many fields
    - Each year there is at least a new paper on the topic
  - It's a really good way to look at and learn many different issues with a grid topology
- Let's look at the natural matrix distribution scheme induced by a grid/torus

Courtesy of Henri Casanova

2-D Matrix Distribution



- We denote by $a_{i,j}$ an element of the matrix
- We denote by $A_{i,j}$ (or A_{ij}) the block of the matrix allocated to $P_{i,j}$

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Courtesy of Henri Casanova

How do Matrices Get Distributed? (Sec. 4.7)

- Data distribution can be completely ad-hoc
- But what about when developing a library that will be used by others?
- There are two main options:
- Centralized
  - when calling a function (e.g., matrix multiplication)
    - the input data is available on a single “master” machine (perhaps in a file)
    - the input data must then be distributed among workers
    - the output data must be undistributed and returned to the “master” machine (perhaps in a file)
  - More natural/easy for the user
  - Allows for the library to make data distribution decisions transparently to the user
  - Prohibitively expensive if one does sequences of operations
    - and one almost always does so
- Distributed
  - when calling a function (e.g., matrix multiplication)
    - Assume that the input is already distributed
    - Leave the output distributed
  - May lead to having to “redistribute” data in between calls so that distributions match, which is harder for the user and may be costly as well
    - For instance one may want to change the block size between calls, or go from a non-cyclic to a cyclic distribution
- Most current software adopt the distributed approach
  - more work for the user
  - more flexibility and control
- We’ll always assume that the data is magically already distributed by the user

Courtesy of Henri Casanova

Four Matrix Multiplication Algorithms

- We'll look at four algorithms
  - Outer-Product
  - Cannon
  - Fox
  - Snyder
- The first one is used in practice
- The other three are more "historical" but are really interesting to discuss
  - We'll have a somewhat hand-wavy discussion here, rather than look at very detailed code

Courtesy of Henri Casanova

The Outer-Product Algorithm

- Consider the "natural" sequential matrix multiplication algorithm

```
for i=0 to n-1
    for j=0 to n-1
        for k=0 to n-1
            $c_{i,j} += a_{i,k} * b_{k,j}$
```

  - This algorithm is a sequence of inner-products (also called scalar products)
- We have seen that we can switch loops around
- Let's consider this version

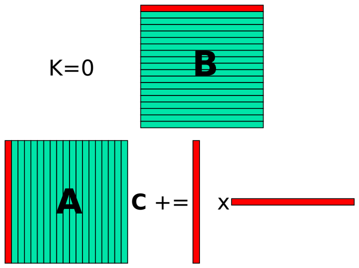
```
for k=0 to n-1
    for i=0 to n-1
        for j=0 to n-1
            $c_{i,j} += a_{i,k} * b_{k,j}$
```

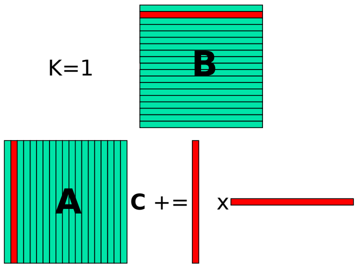
  - This algorithm is a sequence of outer-products!

Courtesy of Henri Casanova

The Outer-Product Algorithm

```
for k=0 to n-1
    for i=0 to n-1
        for j=0 to n-1
            $c_{i,j}$ += $a_{i,k}$ * $b_{k,j}$
```





Courtesy of Henri Casanova

The outer-product algorithm

- Why do we care about switching the loops around to view the matrix multiplication as a sequence of outer products?
- Because it makes it possible to design a very simple parallel algorithm on a grid of processors!
- First step: view the algorithm in terms of the blocks assigned to the processors

```
for k=0 to q-1
    for i=0 to q-1
        for j=0 to q-1
            $C_{i,j} += A_{i,k} * B_{k,j}$
```

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Courtesy of Henri Casanova

The Outer-Product Algorithm

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

$$\begin{array}{cccc}
B_0 & B_0 & B_0 & B_0 \\
B_1 & B_1 & B_1 & B_1 \\
B_2 & B_2 & B_2 & B_2 \\
B_3 & B_3 & B_3 & B_3 \\
0 & 1 & 2 & 3
\end{array}$$

```
for k=0 to q-1
    for i=0 to q-1
        for j=0 to q-1
            $C_{i,j} += A_{i,k} * B_{k,j}$
```

- At step k, processor $P_{i,j}$ needs $A_{i,k}$ and $B_{k,j}$
  - If k = j, then the processor already has the needed block of A
    - Otherwise, it needs to get it from $P_{i,k}$
  - If k = I, then the processor already has the needed block of B
    - Otherwise, it needs to get it from $P_{k,j}$

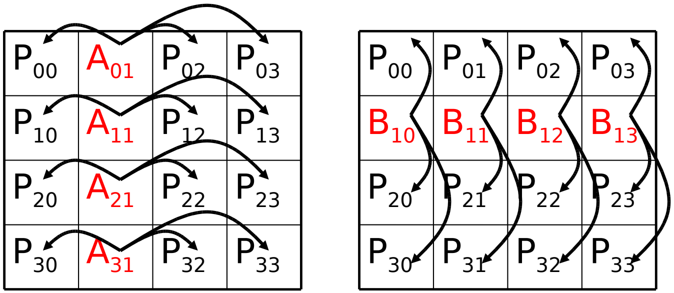
Courtesy of Henri Casanova

The Outer-Product Algorithm

- Based on the previous statements, we can now see how the algorithm works
- At step k
  - Processor $P_{i,k}$ broadcasts its block of matrix A to all processors in processor row i
    - True for all i
  - Processor $P_{k,j}$ broadcasts its block of matrix B to all processor in processor column j
    - True for all j
- There are q-1 steps

Courtesy of Henri Casanova

The Outer Product Algorithm


P_{00} A_{01} P_{02} P_{03}
P_{10} A_{11} P_{12} P_{13}
P_{20} A_{21} P_{22} P_{23}
P_{30} A_{31} P_{32} P_{33}
P_{00} P_{01} P_{02} P_{03}
B_{10} B_{11} B_{12} B_{13}
P_{20} P_{21} P_{22} P_{23}
P_{30} P_{31} P_{32} P_{33}

Step k=1 of the algorithm

Courtesy of Henri Casanova

The Outer-Product Algorithm

```
// m = n/q
var A, B, C: array[0..m-1, 0..m-1] of real
var bufferA, bufferB: array[0..m-1, 0..m-1] of real
var myrow, mycol
myrow = **My_Proc_Row**()
mycol = **My_Proc_Col**()
for k = 0 to q-1
    // Broadcast A along rows
    for i = 0 to q-1
        BroadcastRow(i,k,A,bufferA,m*m)
    // Broadcast B along columns
    for j=0 to q-1
        BroadcastCol(k,j,B,bufferB,m*m)
    // Multiply Matrix blocks (assuming a convenient MatrixMultiplyAdd()
    function)
    if (myrow == k) and (mycol == k)
        MatrixMultiplyAdd(C,A,B,m)
    else if (myrow == k)
        MatrixMultiplyAdd(C,bufferA,B,m)
    else if (mycol == k)
        MatrixMultiplyAdd(C, A, bufferB, m)
    else
        MatrixMultiplyAdd(C, bufferA, bufferB, m)
```

Courtesy of Henri Casanova

Performance Analysis

- The performance analysis is straightforward
- With a one-port model:
  - The matrix multiplication at step k can occur in parallel with the broadcasts at step k+1
  - Both broadcasts happen in sequence
  - Therefore, the execution time is equal to:

$T(m,q) = 2\ Tbcast + (q-1) \max(2\ Tbcast, m^3 w) + m^3 w$

  - w: elementary += * operation
  - Tbcast: time necessary for the broadcast
- With a multi-port model:
  - Both broadcasts can happen at the same time

$T(m,q) = Tbcast + (q-1) \max(Tbcast, m^3 w) + m^3 w$

- The time for a broadcast, using the pipelined broadcast:

  $Tbcast = (\mathrm{sqrt}((q-2)L) + \mathrm{sqrt}(m^2 b))^2$
- When n gets large: $T(m,q) \sim q\, m^3 = n^3 / q^2$
- Thus, asymptotic parallel efficiency is 1!

Courtesy of Henri Casanova

So what?

- On a ring platform we had already given an asymptotically optimal matrix multiplication algorithm on a ring in an earlier set of slides
- So what's the big deal about another asymptotically optimal algorithm?
- Once again, when n is huge, indeed we don't care
- But communication costs are often non-negligible and do matter
  - When n is "moderate"
  - When w/b is low
- It turns out, that the grid topology is advantageous for reducing communication costs!

Courtesy of Henri Casanova

Ring vs. Grid

- When we discussed the ring, we found that the communication cost of the matrix multiplication algorithm was: n^2 b
  - A each step, the algorithm sends n^2/p matrix elements among neighboring processors
  - There are p steps
- For the algorithm on a grid:
  - Each step involves 2 broadcasts of n^2/p matrix elements
    - Assuming a one-port model, not to give an “unfair” advantage to the grid topology
  - Using a pipelined broadcast, this can be done in approximately the same time as sending n^2/p matrix elements between neighboring processors on each ring (unless n is really small)
  - Therefore, at each step, the algorithm on a grid spends twice as much time communicating as the algorithm on a ring
  - But it does sqrt(p) fewer steps!
- **Conclusion**: the algorithm on a grid spends at least sqrt(p) less time in communication than the algorithm on a ring

Courtesy of Henri Casanova

Grid vs. Ring

- Why was the algorithm on a Grid much better?
- Reason: More communication links can be used in parallel
  - Point-to-point communication replaced by broadcasts
  - Horizontal and vertical communications may be concurrent
  - More network links used at each step
- Of course, this advantage isn't really an advantage if the underlying physical platform does not really look like a grid
- But, it turns out that the 2-D distribution is inherently superior to the 1-D distribution, no matter what the underlying platform is!

Courtesy of Henri Casanova

Grid vs. Ring

- On a ring
  - The algorithm communicates p matrix block rows that each contain n^2/p elements, p times
  - Total number of elements communicated: pn^2
- On a grid
  - Each step, 2sqrt(p) blocks of n^2/p elements are sent, each to sqrt(p)-1 processors, sqrt(p) times
  - Total number of elements communicated: $2\text{sqrt}(p)n^2$
- Conclusion: the algorithm with a grid in mind inherently sends less data around than the algorithm on a ring
- Using a 2-D data distribution would be better than using a 1-D data distribution even if the underlying platform were a non-switched Ethernet for instance!
  - Which is really 1 network link, and one may argue is closer to a ring (p comm links) than a grid (p^2 comm links)

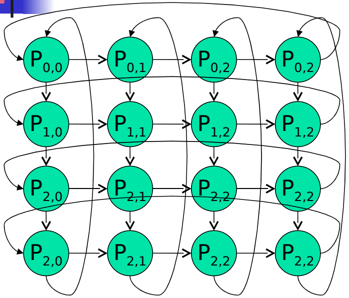
Courtesy of Henri Casanova

Conclusion

- Writing algorithms on a grid topology is a little bit more complicated than in a ring topology
- But there is often a payoff in practice and grid topologies are very popular

Courtesy of Henri Casanova

2-D Matrix Distribution



- We denote by $a_{i,j}$ an element of the matrix
- We denote by $A_{i,j}$ (or A_{ij}) the block of the matrix allocated to $P_{i,j}$

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Courtesy of Henri Casanova

The Cannon Algorithm

- This is a very old algorithm
  - From the time of systolic arrays
  - Adapted to a 2-D grid
- The algorithm starts with a redistribution of matrices A and B
  - Called "preskewing"
- Then the matrices are multiplied
- Then the matrices are re-redistributed to match the initial distribution
  - Called "postskewing"

Courtesy of Henri Casanova

Cannon’s Preskewing

- Matrix A: each block row of matrix A is shifted so that each processor in the first processor column holds a diagonal block of the matrix

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{11}	A_{12}	A_{13}	A_{14}
A_{22}	A_{23}	A_{20}	A_{21}
A_{33}	A_{30}	A_{31}	A_{32}

Courtesy of Henri Casanova

Cannon’s Preskewing

- Matrix B: each block column of matrix B is shifted so that each processor in the first processor row holds a diagonal block of the matrix

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

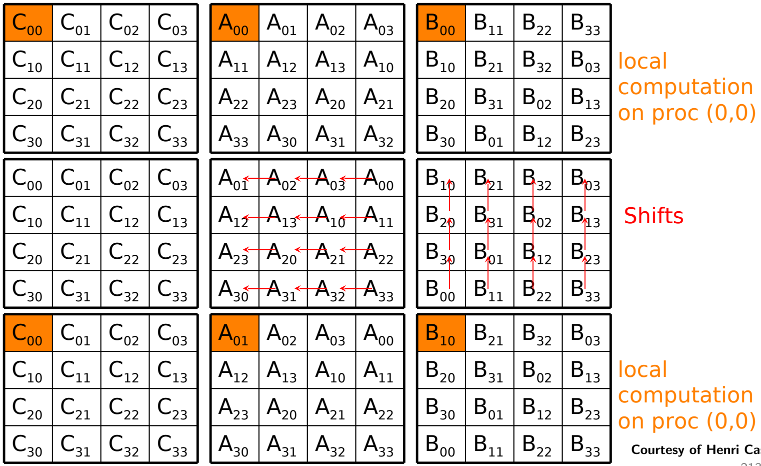
B_{00}	B_{11}	B_{22}	B_{33}
B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}

Cannon’s Computation

- The algorithm proceeds in q steps
- At each step each processor performs the multiplication of its block of A and B and adds the result to its block of C
- Then blocks of A are shifted to the left and blocks of B are shifted upward
  - Blocks of C never move
- Let’s see it on a picture

Courtesy of Henri Casanova

Cannon's Steps


local computation on proc (0,0)
Shifts
local computation on proc (0,0)

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{11}	A_{12}	A_{13}	A_{10}
A_{22}	A_{23}	A_{20}	A_{21}
A_{33}	A_{30}	A_{31}	A_{32}

B_{00}	B_{11}	B_{22}	B_{33}
B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}

local computation on proc (0,0)

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{01}	A_{02}	A_{03}	A_{00}
A_{12}	A_{13}	A_{10}	A_{11}
A_{23}	A_{20}	A_{21}	A_{22}
A_{30}	A_{31}	A_{32}	A_{33}

B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}
B_{00}	B_{11}	B_{22}	B_{33}

Shifts

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{01}	A_{02}	A_{03}	A_{00}
A_{12}	A_{13}	A_{10}	A_{11}
A_{23}	A_{20}	A_{21}	A_{22}
A_{30}	A_{31}	A_{32}	A_{33}

B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}
B_{00}	B_{11}	B_{22}	B_{33}

local computation on proc (0,0)

Courtesy of Henri Casanova

The Algorithm

```
Participate in preskewing of A
Partitipate in preskweing of B
For k = 1 to q
    Local C = C + A*B
    Vertical shift of B
    Horizontal shift of A
Participate in postskewing of A
Partitipate in postskewing of B
```

Courtesy of Henri Casanova

Performance Analysis

- Let’s do a simple performance analysis with a 4-port model
  - The 1-port model is typically more complicated
- Symbols
  - n: size of the matrix
  - qxq: size of the processor grid
  - m = n / q
  - L: communication start-up cost
  - w: time to do a basic computation (+= . * .)
  - b: time to communicate a matrix element
- T(m,q) = Tpreskew + Tcompute + Tpostskew

Courtesy of Henri Casanova

Pre/Post-skewing times

- Let's consider the horizontal shift
- Each row must be shifted so that the diagonal block ends up on the first column
- On a mono-directional ring:
  - The last row needs to be shifted (q-1) times
  - All rows can be shifted in parallel
  - Total time needed: (q-1) (L + m^2 b)
- On a bi-directional ring, a row can be shifted left or right, depending on which way is shortest!
  - A row is shifted at most floor(q/2) times
  - All rows can be shifted in parallel
  - Total time needed: floor(q/2) (L + m^2 b)
- Because of the 4-port assumption, preskewing of A and B can occur in parallel (horizontal and vertical shifts do not interfere)
- Therefore: Tpreskew = Tpostskew = floor(q/2) (L+m^{2}b)

Courtesy of Henri Casanova

Time for each step

- At each step, each processor computes an mxm matrix multiplication
  - Compute time: m^3 w
- At each step, each processor sends/receives a mxm block in its processor row and its processor column
  - Both can occur simultaneously with a 4-port model
  - Takes time L+ m^2b
- Therefore, the total time for the q steps is: Tcompute = q max (L + m^2b, m^3w)

Courtesy of Henri Casanova

Cannon Performance Model

- $T(m,n) = 2 * \text{floor}(q/2)\ (L + m^2 b) + q \max(m^3 w, L + m^2 b)$
- This performance model is easily adapted
  - If one assumes mono-directional links, then the “floor(q/2)” above becomes “(q-1)”
  - If one assumes 1-port, there is a factor 2 added in front of communication terms
  - If one assumes no overlap of communication and computation at a

Courtesy of Henri Casanova

The Fox Algorithm

- This algorithm was originally developed to run on a hypercube topology
  - But in fact it uses a grid, embedded in the hypercube
- This algorithm requires no pre- or post-skewing
- It relies on horizontal broadcasts of the diagonals of matrix A and on vertical shifts of matrix B
- Sometimes called the "multiply-broadcast-roll" algorithm
- Let's see it on a picture
  - Although it's a bit awkward to draw because of

Courtesy of Henri Casanova

Execution Steps...

initial state

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Broadcast of A's 1st diag. (stored in a Separate buffer)

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{00}	A_{00}	A_{00}
A_{11}	A_{11}	A_{11}	A_{11}
A_{22}	A_{22}	A_{22}	A_{22}
A_{33}	A_{33}	A_{33}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Local computation

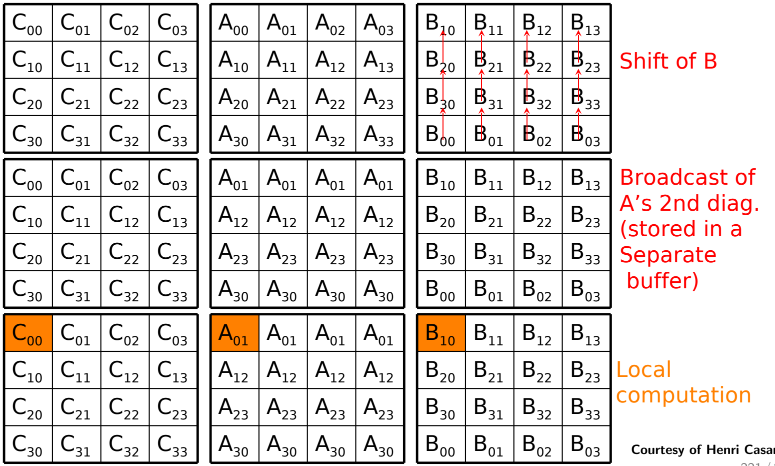
C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{00}	A_{00}	A_{00}
A_{11}	A_{11}	A_{11}	A_{11}
A_{22}	A_{22}	A_{22}	A_{22}
A_{33}	A_{33}	A_{33}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Courtesy of Henri Casanova

Execution Steps...



C_{00}	C_{01}	C_{02}	C_{03}	A_{00}	A_{01}	A_{02}	A_{03}	B_{10}	B_{11}	B_{12}	B_{13}
C_{10}	C_{11}	C_{12}	C_{13}	A_{10}	A_{11}	A_{12}	A_{13}	B_{20}	B_{21}	B_{22}	B_{23}
C_{20}	C_{21}	C_{22}	C_{23}	A_{20}	A_{21}	A_{22}	A_{23}	B_{30}	B_{31}	B_{32}	B_{33}
C_{30}	C_{31}	C_{32}	C_{33}	A_{30}	A_{31}	A_{32}	A_{33}	B_{00}	B_{01}	B_{02}	B_{03}

C_{00}	C_{01}	C_{02}	C_{03}	A_{01}	A_{01}	A_{01}	A_{01}	B_{10}	B_{11}	B_{12}	B_{13}
C_{10}	C_{11}	C_{12}	C_{13}	A_{12}	A_{12}	A_{12}	A_{12}	B_{20}	B_{21}	B_{22}	B_{23}
C_{20}	C_{21}	C_{22}	C_{23}	A_{23}	A_{23}	A_{23}	A_{23}	B_{30}	B_{31}	B_{32}	B_{33}
C_{30}	C_{31}	C_{32}	C_{33}	A_{30}	A_{30}	A_{30}	A_{30}	B_{00}	B_{01}	B_{02}	B_{03}

C_{00}	C_{01}	C_{02}	C_{03}	A_{01}	A_{01}	A_{01}	A_{01}	B_{10}	B_{11}	B_{12}	B_{13}
C_{10}	C_{11}	C_{12}	C_{13}	A_{12}	A_{12}	A_{12}	A_{12}	B_{20}	B_{21}	B_{22}	B_{23}
C_{20}	C_{21}	C_{22}	C_{23}	A_{23}	A_{23}	A_{23}	A_{23}	B_{30}	B_{31}	B_{32}	B_{33}
C_{30}	C_{31}	C_{32}	C_{33}	A_{30}	A_{30}	A_{30}	A_{30}	B_{00}	B_{01}	B_{02}	B_{03}

Courtesy of Henri Casanova

Fox's Algorithm

```
// No initial data movement
for k = 1 to q  in parallel
  Broadcast A's $k^{th}$ diagonal
  Local C = C + A*B
  Vertical shift of B
// No final data movement
```

- Again note that there is an additional array to store incoming diagonal block
- This is the array we use in the A*B multiplication

Courtesy of Henri Casanova

Performance Analysis

- You'll have to do it in a homework assignment
  - Write pseudo-code of the algorithm in more details
  - Write the performance analysis

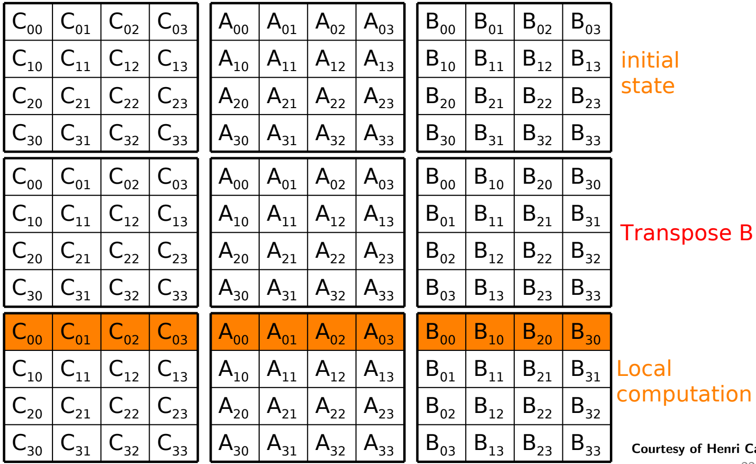
Courtesy of Henri Casanova

Snyder’s Algorithm (1992)

- More complex than Cannon’s or Fox’s
- First transposes matrix B
- Uses reduction operations (sums) on the rows of matrix C
- Shifts matrix B

Courtesy of Henri Casanova

Execution Steps...



initial state

C_{00}	C_{01}	C_{02}	C_{03}	A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{01}	B_{02}	B_{03}
C_{10}	C_{11}	C_{12}	C_{13}	A_{10}	A_{11}	A_{12}	A_{13}	B_{10}	B_{11}	B_{12}	B_{13}
C_{20}	C_{21}	C_{22}	C_{23}	A_{20}	A_{21}	A_{22}	A_{23}	B_{20}	B_{21}	B_{22}	B_{23}
C_{30}	C_{31}	C_{32}	C_{33}	A_{30}	A_{31}	A_{32}	A_{33}	B_{30}	B_{31}	B_{32}	B_{33}

Transpose B

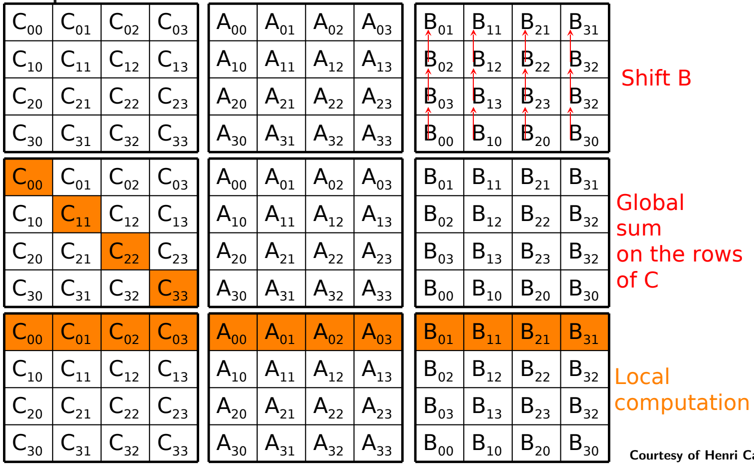
C_{00}	C_{01}	C_{02}	C_{03}	A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{10}	B_{20}	B_{30}
C_{10}	C_{11}	C_{12}	C_{13}	A_{10}	A_{11}	A_{12}	A_{13}	B_{01}	B_{11}	B_{21}	B_{31}
C_{20}	C_{21}	C_{22}	C_{23}	A_{20}	A_{21}	A_{22}	A_{23}	B_{02}	B_{12}	B_{22}	B_{32}
C_{30}	C_{31}	C_{32}	C_{33}	A_{30}	A_{31}	A_{32}	A_{33}	B_{03}	B_{13}	B_{23}	B_{33}

Local computation

🟧 C_{00}	🟧 C_{01}	🟧 C_{02}	🟧 C_{03}	🟧 A_{00}	🟧 A_{01}	🟧 A_{02}	🟧 A_{03}	🟧 B_{00}	🟧 B_{10}	🟧 B_{20}	🟧 B_{30}
C_{10}	C_{11}	C_{12}	C_{13}	A_{10}	A_{11}	A_{12}	A_{13}	B_{01}	B_{11}	B_{21}	B_{31}
C_{20}	C_{21}	C_{22}	C_{23}	A_{20}	A_{21}	A_{22}	A_{23}	B_{02}	B_{12}	B_{22}	B_{32}
C_{30}	C_{31}	C_{32}	C_{33}	A_{30}	A_{31}	A_{32}	A_{33}	B_{03}	B_{13}	B_{23}	B_{33}

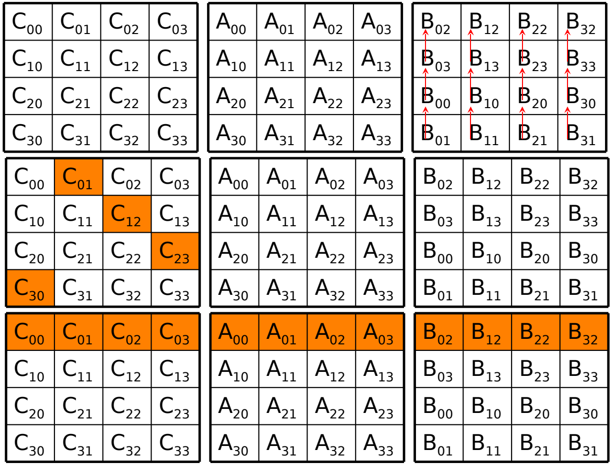
Courtesy of Henri Casanova

Execution Steps...


C_{00} C_{01} C_{02} C_{03}
C_{10} C_{11} C_{12} C_{13}
C_{20} C_{21} C_{22} C_{23}
C_{30} C_{31} C_{32} C_{33}
A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}
B_{01} B_{11} B_{21} B_{31}
B_{02} B_{12} B_{22} B_{32}
B_{03} B_{13} B_{23} B_{32}
B_{00} B_{10} B_{20} B_{30}
Shift B
Global sum on the rows of C
Local computation

Courtesy of Henri Casanova

Execution Steps...


C_{00} C_{01} C_{02} C_{03}
C_{10} C_{11} C_{12} C_{13}
C_{20} C_{21} C_{22} C_{23}
C_{30} C_{31} C_{32} C_{33}
A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}
B_{02} B_{12} B_{22} B_{32}
B_{03} B_{13} B_{23} B_{33}
B_{00} B_{10} B_{20} B_{30}
B_{01} B_{11} B_{21} B_{31}

Shift B

Global sum on the rows of C

Local computation

Courtesy of Henri Casanova

The Algorithm

```
var A,B,C: array[0..m-1][0..m-1] of real
var bufferC: array[0..m-1][0..m-1] of real
Transpose B
MatrixMultiplyAdd(bufferC, A, B, m)
Vertical shifts of B
For k = 1 to q-1
    Global sum of bufferC on proc rows into $C_{i,(i+k-1)\%q}$
    MatrixMultiplyAdd(bufferC, A, B, m)
    Vertical shift of B
Global sum of bufferC on proc rows into $C_{i,(i+k-1)\%q}$
Transpose B
```

Courtesy of Henri Casanova

Performance Analysis

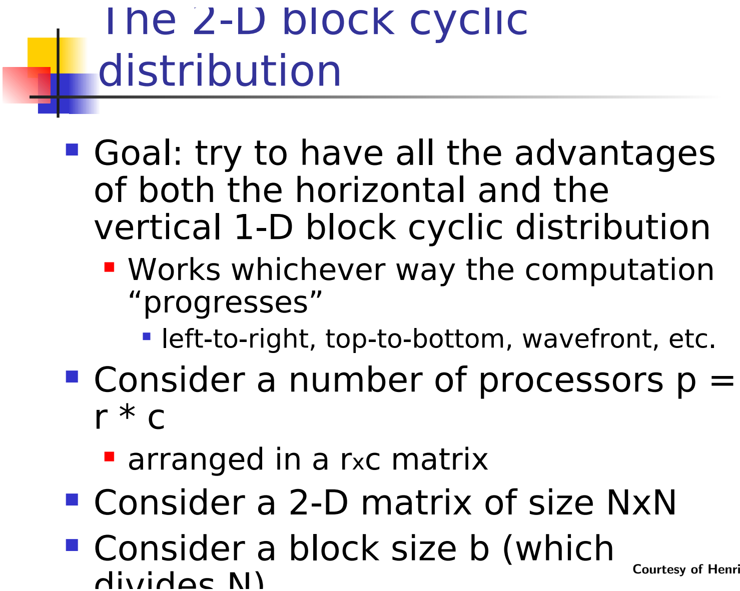
- The performance analysis isn’t fundamentally different than what we’ve done so far
- But it’s a bit cumbersome
- See the textbook
  - in particular the description of the matrix transposition (see also Exercise 5.1)

Courtesy of Henri Casanova

Which Data Distribution?

- So far we've seen:
  - Block Distributions
  - 1-D Distributions
  - 2-D Distributions
  - Cyclic Distributions
- One may wonder what a good choice is for a data distribution?
- Many people argue that a good "Swiss Army knife" is the "2-D block cyclic distribution

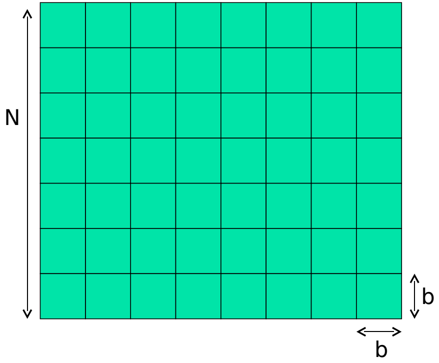
Courtesy of Henri Casanova



The 2-D block cyclic distribution

- Goal: try to have all the advantages of both the horizontal and the vertical 1-D block cyclic distribution
  - Works whichever way the computation "progresses"
    - left-to-right, top-to-bottom, wavefront, etc.
- Consider a number of processors p = r * c
  - arranged in a rxc matrix
- Consider a 2-D matrix of size NxN
- Consider a block size b (which divides N)

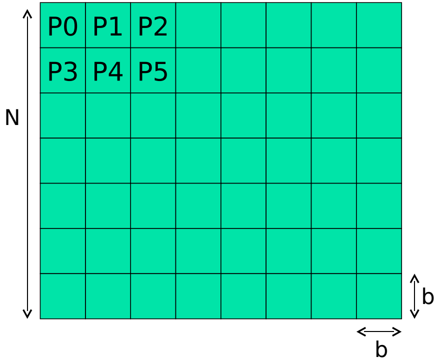
The 2-D block cyclic distribution

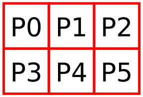


P0	P1	P2
P3	P4	P5

Courtesy of Henri Casanova

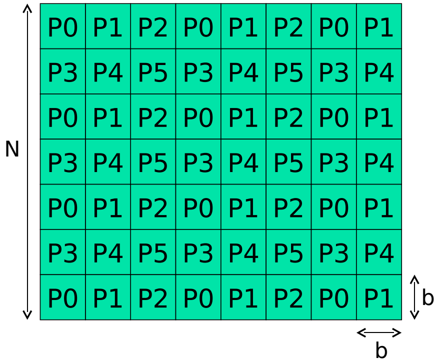
The 2-D block cyclic distribution





Courtesy of Henri Casanova

The 2-D block cyclic distribution



P0	P1	P2
P3	P4	P5

- Slight load imbalance
  - Becomes negligible with many blocks
- Index computations had better be implemented in separate functions
- Also: functions that tell a process who its neighbors are
- Overall, requires a whole infrastructure, but many think you can't go wrong with this distribution

Courtesy of Henri Casanova

Conclusion

- All the algorithms we have seen in the semester can be implemented on a 2-D block cyclic distribution
- The code ends up much more complicated
- But one may expect several benefits “for free”
- The ScaLAPAK library recommends to use the 2-D block cyclic distribution
  - Although its routines support all other distributions

Courtesy of Henri Casanova

A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman.
LogGP: Incorporating long messages into the LogP model for parallel computation.
Journal of Parallel and Distributed Computing, 44(1):71–79, 1997.

D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken.
LogP: a practical model of parallel computation.
Communication of the ACM, 39(11):78–85, 1996.

R. W. Hockney.
The communication challenge for mpp : Intel paragon and meiko cs-2.
Parallel Computing, 20:389–398, 1994.

B. Hong and V.K. Prasanna.
Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput.

In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

T. Kielmann, H. E. Bal, and K. Verstoep.
Fast measurement of LogP parameters for message passing platforms.
In *Proceedings of the 15th IPDPS. Workshops on Parallel and Distributed Processing*, 2000.

Steven H. Low.
A duality model of TCP and queue management algorithms.
IEEE/ACM Transactions on Networking, 2003.

Dong Lu, Yi Qiao, Peter A. Dinda, and Fabián E. Bustamante.
Characterizing and predicting tcp throughput on the wide area network.
In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien.
Mapping and load-balancing iterative computations on heterogeneous clusters with shared links.
IEEE Trans. Parallel Distributed Systems, 15(6):546–558, 2004.

Maxime Martinasso.
Analyse et modélisation des communications concurrentes dans les réseaux haute performance.
PhD thesis, Université Joseph Fourier de Grenoble, 2007.

Laurent Massoulié and James Roberts.
Bandwidth sharing: Objectives and algorithms.
In *INFOCOM (3)*, pages 1395–1403, 1999.

Loris Marchal, Yang Yang, Henri Casanova, and Yves Robert.
Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms.

Int. Journal of High Performance Computing Applications, (3), 2006.

Frédéric Wagner.
Redistribution de données à travers un réseau haut débit.
PhD thesis, Université Henri Poincaré Nancy 1, 2005.