Communications on Distributed Architectures

Arnaud LEGRAND, CR CNRS, LIG/INRIA/Mescal

Vincent DANJEAN, MCF UJF, LIG/INRIA/Moais

October, 11th 2010

Goals of this lecture

Understand how communication libraries can efficiently use high speed networks

Understand the difficulties to write efficient parallel programs targeting several architectures.

High Performance Networking

1. Current high speed network characteristics
   - (Fast|Giga)-Ethernet
   - Legacy hardware
   - Current hardware
2. Classical techniques for efficient communications
   - Interacting with the network card: PIO and DMA
   - Zero-copy communications
   - Handshake Protocol
   - OS Bypass
3. Some low-level interfaces
   - BIP and MX/Myrinet
   - SiSCI/SCI
   - VIA
4. Summary

Portability and Efficiency

5. Optimizing communications
   - Optimizing communication methods
   - An experimental project: the Madeleine interface
6. Asynchronous communications
   - MPI example
   - Mixing threads and communications
7. Hierarchical plate-forms and efficient scheduling
   - Programming on current SMP machines
   - BubbleSched: guiding scheduling through bubbles
8. Conclusion
   - High-performance parallel programming is difficult

Part I

High Performance Networking

Outlines

High Speed Networks

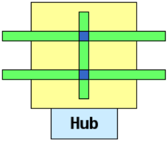
High Speed Networks are used in clusters

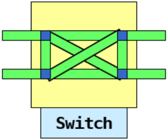
- low distance
- very interesting performance
  - low latency: about 1 μs
  - high bandwidth: about 10 Gb/s and more
- specific light protocols
  - static routing of messages
  - no required packet fragmentation
  - sometimes, no packet required

Myrinet, Quadrics, SCI, ...

(Fast|Giga)-Ethernet

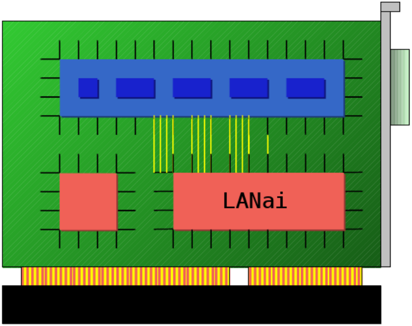
- Interconnect:
  - Hub or switch
- Wires:
  - Copper or optical fiber
- Latency:
  - about 10 μs
- Bandwidth:
  - From 100 Mb/s to 10 Gb/s (100 Gb/s, june 2010)
- Remark:
  - compatible with traditional Ethernet





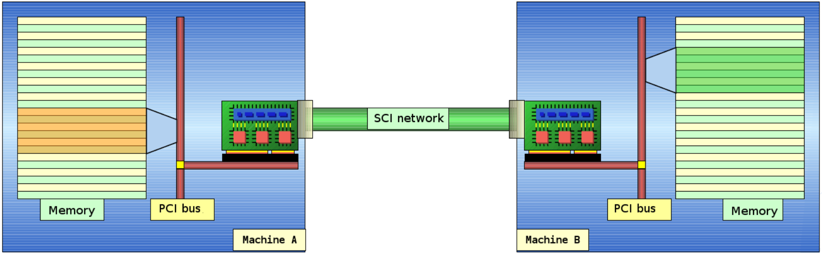
Myrinet

- Myricom corporate
- Interconnect:
  - Switch
- PCI card with:
  - a processor: LANai
  - SRAM memory: about 4 MB
- Latency:
  - about 1 or 2 μs
- Bandwidth:
  - 10 Gb/s
- Remark:
  - static, wormhole routing
  - can you RJ45 cables



SCI

- Scalable Coherent Interface
  - IEEE norm (1993)
  - Dolphin corporate
- Uses remote memory access:
  - Address space remotely mapped



InfiniBand

- Several manufacturers (Cisco, HP, Intel, IBM, etc.)
- Interconnect:
  - Optical links
  - Serial, point-to-point connections
  - Switched fabric (possibility of several paths)
- Bandwidth
  - single line of 2, 4, 8, 14 or 25 Mb/s
  - possibility of bonding 4 or 12 lines
- Latency:
  - about 100 or 200 ns *for hardware only*
  - about 1 or 2 μs for some hardware with its driver
- Remark:
  - can interconnect buildings
  - RDMA operations available

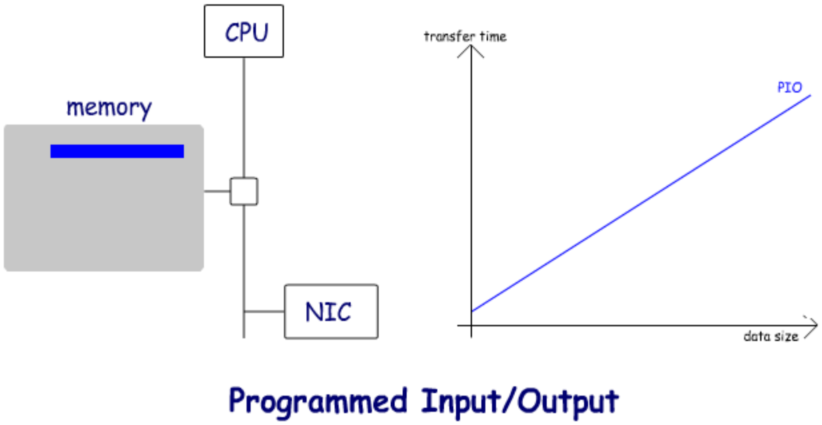
Quadrics

- One manufacturer (Quadrics)
- Interconnect:
  - Bi-directional serial links
  - Switched fabric (possibility of several paths)
- Bandwidth
  - 1 to 2 Gb/s on each direction
- Latency:
  - about 1.3 μs in MPI
- Remark:
  - selected by Bull for the fastest supercomputer in Europe: Tera100 at CEA
  - global operations (reduction, barrier) available in hardware

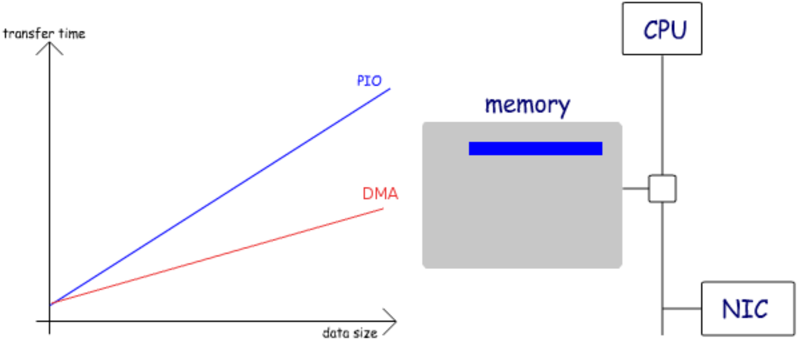
Outlines

1. Current high speed network characteristics
2. Classical techniques for efficient communications
    - Interacting with the network card: PIO and DMA
    - Zero-copy communications
    - Handshake Protocol
    - OS Bypass
3. Some low-level interfaces
4. Summary

Interacting with the network card: PIO mode



Interacting with the network card: DMA mode



Direct Memory Access

Zero-copy communications

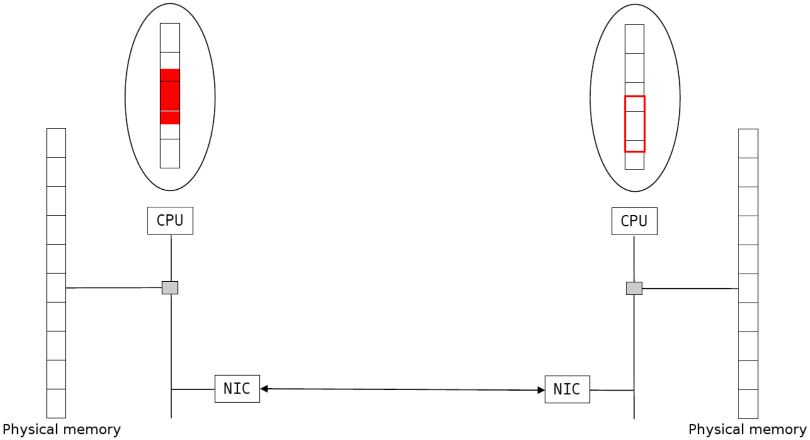
Goals

- Reduce the communication time
  - Copy time cannot be neglected
    - but it can be partially recovered with pipelining
- Reduce the processor use
  - currently, `memcpy` are executed by processor instructions

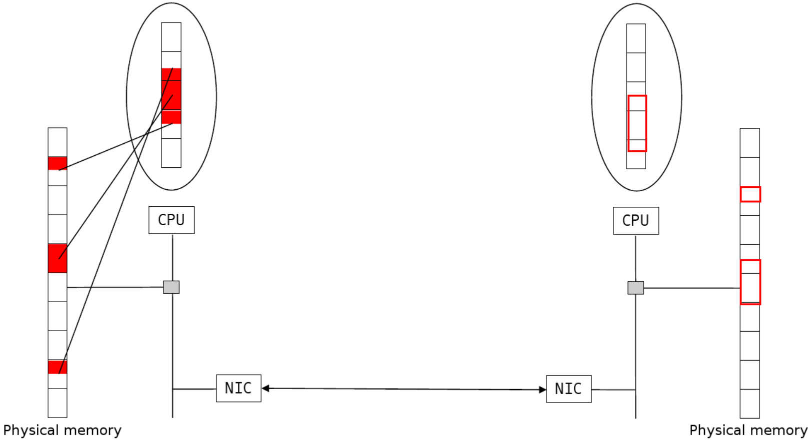
Idea

The network card directly read/write data from/to the application memory

Zero-copy communications



Zero-copy communications



Zero-copy communications for emission

PIO mode transfers

- No problem for zero-copy

DMA mode transfers

- Non contiguous data in physical memory
- Headers added in the protocol
  - linked DMA
  - limits on the number of non contiguous segments

Zero-copy communications for reception

A network card cannot “freeze” the received message on the physical media

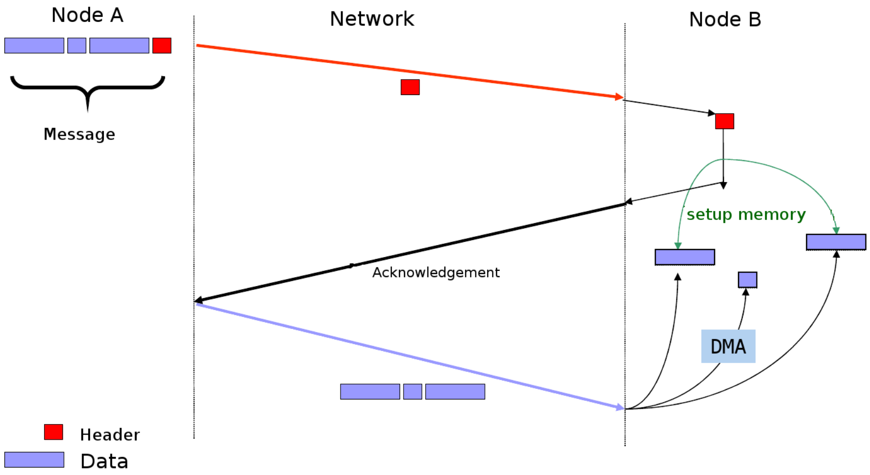
If the receiver posted a “recv” operation before the message arrives

- zero-copy OK if the card can filter received messages
- else, zero-copy allowed with bounded-sized messages with optimistic heuristics

If the receiver is not ready

- A handshake protocol must be setup for big messages
- Small messages can be stored in an internal buffer

Using a Handshake Protocol



A few more considerations

The receiving side plays an important role

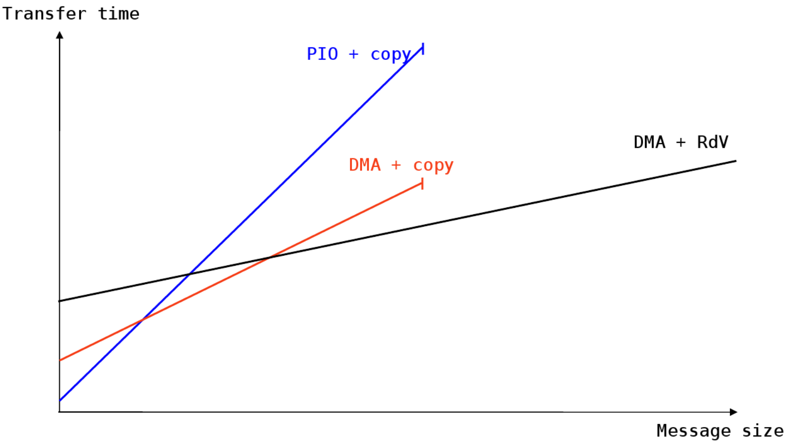
- Flow-control is mandatory
- Zero-copy transfers
  - the sender has to ensure that the receiver is ready
  - a handshake (REQ+ACK) can be used

Communications in user-space introduce some difficulties

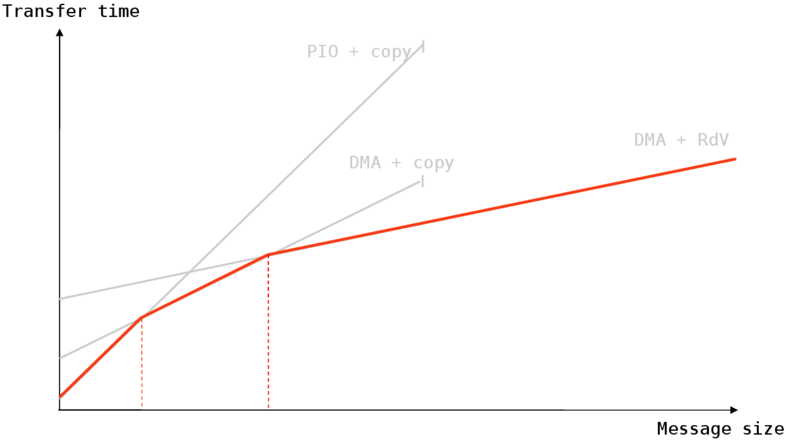
- Direct access to the NIC
  - most technologies impose “pinned” memory pages

Network drivers have limitations

Communication Protocol Selection

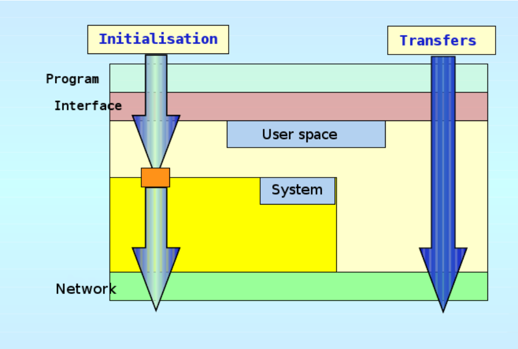


Communication Protocol Selection



Operating System Bypass

- Initialization
  - traditional system calls
  - only at session beginning
- Transfers
  - direct from user space
  - no system call
  - “less” interrupts
- Humm. . . And what about security ?

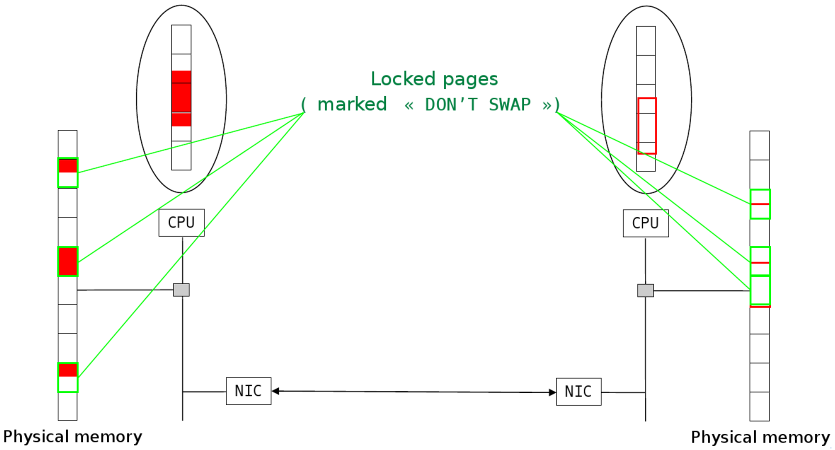


OS-bypass + zero-copy

Problem

- Zero-copy mechanism uses DMA that requires physical addresses
- Mapping between virtual and physical address is only known by:
  - the processor (MMU)
  - the OS (pages table)
- We need that
  - the library knows this mapping
  - this mapping is not modified during the communication
    - ex: swap decided by the OS, copy-on-write, etc.
- No way to ensure this in user space !

OS-bypass + zero-copy



OS-bypass + zero-copy

First solution

- Pages “recorded” in the kernel to avoid swapping
- Management of a cache for virtual/physical addresses mapping
  - in user space or on the network card
- Diversion of system calls that can modify the address space

Second solution

- Management of a cache for virtual/physical addresses mapping on the network card
- OS patch so that the network card is “advertised” when a modification occurs
- Solution chosen by MX/Myrinet and Elan/Quadrics

Direct consequences

- Latency measure can vary whether the memory region used
  - Some pages are “recorded” within the network card
- Ideal case are ping-pong exchanges
  - The same pages are reused hundred of times
- Worst case are applications using lots of different data regions. . .

Outlines

BIP/Myrinet

- Basic Interface for Parallelism
  - L. Prylli and B. Tourancheau
- Dedicated to Myrinet networks
- Characteristics
  - Asynchronous communication
  - No error detection
  - No flow control
    - Small messages are copied into a fixed buffer at reception
    - Big messages are lost if the receiver is not ready

MX/Myrinet

- Myrinet eXpress
  - Official driver from Myricom
- Very simplistic interface to allow easy implementation of MPI
  - Flow control
  - Reliable communications
  - Non contiguous messages
  - Multiplexing

SiSCI/SCI

- Driver for SCI cards
- Programming model
  - Remote memory access
    - Explicit: RDMA
    - Implicit: memory projections
- Performance
  - Explicit use of some operation required:
    - memory “flush”
    - `SCI_memcpy`
    - RDMA

VIA

- Virtual Interface Architecture
- A new standard
  - Lots of industrials
    - Microsoft, Intel, Compaq, etc.
  - Use for InfiniBand networks
- Characteristics
  - Virtual interfaces objects
    - Queues of descriptors (for sending and receiving)
  - Explicit memory recording
  - Remote reads/writes
    - RDMA

Outlines

Summary

Efficient hardware

- very low latency and high bandwidth
- complex hardware to be programmed efficiently
  - onboard CPU, onboard MMU for DMA, etc.

Very specific programming interfaces

- dedicated to specific technologies (but VIA)
- different programming models
- quasi no portability

It is not reasonable to program a scientific application directly with such programming interfaces

Part II

Portability and Efficiency

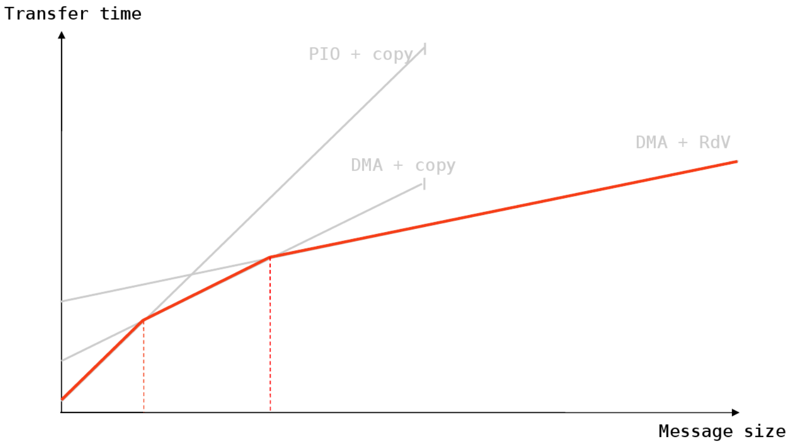
Outlines

Optimizing communication methods

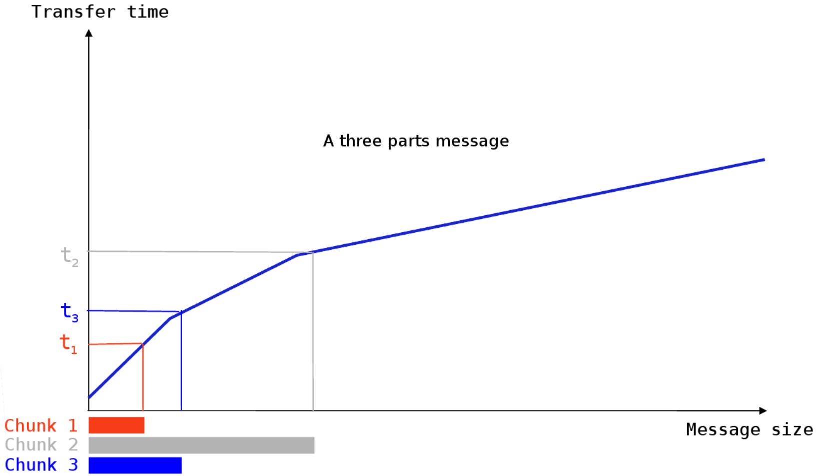
Low-level libraries sometimes prefer using the processor in order to guaranty low latencies

- Depending on the message size
  - PIO for small messages
  - Pipelined copies with DMA for medium messages
  - Zero-copy + DMA for large messages
- Example: limit medium/large is set to 32 KB for MX
  - sending messages from 0 to 32 KB cannot overlap computations

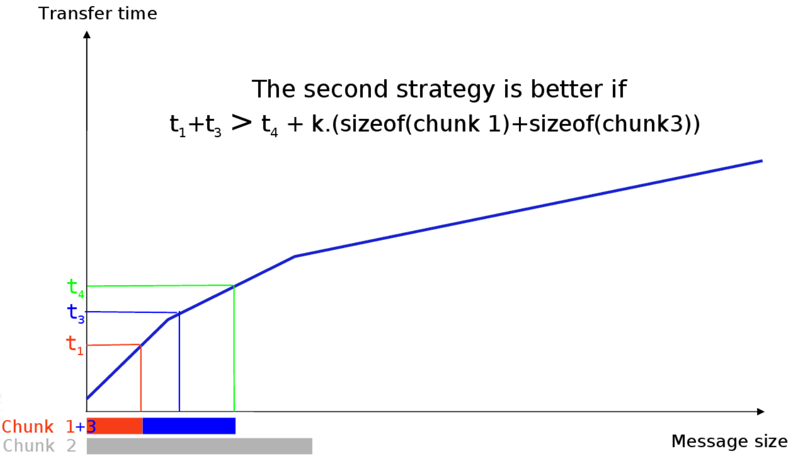
Choosing the Optimal Strategy



Choosing the Optimal Strategy



Choosing the Optimal Strategy



Choosing the Optimal Strategy

It depends on

- The underlying network with driver performance
  - latency
  - PIO and DMA performance
  - Gather/Scatter feature
  - Remote DMA feature
  - etc.
- Multiple network cards ?

But also on

- memory copy performance
- I/O bus performance

Efficient AND portable is not easy

An experimental project: the Madeleine interface

Goals

Rich interface to exchange complex message while keeping the portability

Characteristics

- incremental building of messages with internal dependencies specifications
  - the application specify dependencies and constraints (semantics)
  - the middle-ware automatically choice the best strategy
- multi-protocols communications
  - several networks can be used together
- thread-aware library

Message building

Sender

```
begin_send(dest)

pack(&len, sizeof(int))


pack(data, len)





end_send()
```

Receiver

```
begin_recv()

unpack(&len, sizeof(int))

data = malloc(len)
unpack(data, len)





end_recv()
```

Message building

Sender

```
begin_send(dest)

pack(&len, sizeof(int),
  r_express)

pack(data, len,
  r_cheaper)

end_send()
```

Receiver

```
begin_recv()

unpack(&len, sizeof(int),
  r_express)
data = malloc(len)
unpack(data, len,
  r_cheaper)

end_recv()
```

Message building

Sender

```
begin_send(dest)

pack(&len, sizeof(int),
  r_express)

pack(data, len,
  r_cheaper)

pack(data2, len,
  r_cheaper)

end_send()
```

Receiver

```
begin_recv()

unpack(&len, sizeof(int),
  r_express)
data = malloc(len)
unpack(data, len,
  r_cheaper)
data2 = malloc(len)
unpack(data2, len,
  r_cheaper)

end_recv()
```

How to implement optimizations ?

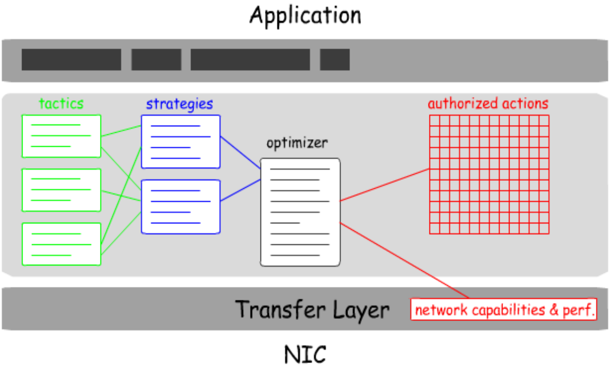
Using parameters and historic

- sender and receiver always take the same (deterministic) decisions
- only data are sent

Using other information

- allow unordered communication (especially for short messages)
  - can required controls messages
- allow dynamically new strategies (plug-ins)
- use “near future”
  - allow small delays or application hints

Optimisations « Just-in-Time »



Courtesy of Olivier Aumage

Why such interfaces ?

Portability of the application

No need to rewrite the application when running on an other kind of network

Efficiency

- local optimizations (aggregation, etc.)
- global optimizations (load-balancing on several networks, etc.)

But non standard interface

rewrite some standard interfaces on top of this one

- some efficiency is lost

Still lots of work

What about

- equity wrt. optimization ?
- finding optimal strategies ?
  - still an open problem in many cases
- convincing users to try theses new interfaces
- managing fault-tolerance
- allowing cluster interconnections (ie high-speed network routing)
- allowing connection and disconnections of nodes
- etc.

Outlines

Message Passing Interface

Characteristics

- Interface (not implementation)
- Different implementations
  - MPICH
  - LAM-MPI
  - OpenMPI
  - and all closed-source MPI dedicated to specific hardware
- MPI 2.0 begins to appear

Several Ways to Exchange Messages with MPI

MPI_Send (standard)

- At the end of the call, data can be reused immediately

MPI_Bsend (buffered)

- The message is locally copied if it cannot be send immediately

MPI_Rsend (ready)

- The sender “promises” that the receiver is ready

MPI_Ssend (synchronous)

- At the end of the call, the reception started (similar to a synchronization barrier)

Non Blocking Primitives

MPI_Isend / MPI_Irecv (immediate)

```
MPI_request r;

MPI_Isend(..., data, len, ..., &r)

// Calculus that does not modify
'data'
MPI_wait(&r, ...);
```

These primitives must be used as much as possible

About MPI Implementations

- MPI is available on nearly all existing networks and protocols!
  - Ethernet, Myrinet, SCI, Quadrics, Infiniband, IP, shared memory, etc.
- MPI implementation are really efficient
  - low latency (hard), large bandwidth (easy)
  - optimized version from hardware manufacturers (IBM, SGI)
  - implementations can be based on low-level interfaces
    - MPICH/Myrinet, MPICH/Quadrics

BUT these “good performance” are often measured with ping-pong programs...

Asynchronous communications with MPI

Token circulation while computing on 4 nodes

```
if (mynode!=0)
  MPI_Recv();

req=MPI_Isend(next);
Work(); /* about 1s */
MPI_Wait(req);

if (mynode==0)
  MPI_Recv();
```

Asynchronous communications with MPI

Token circulation while computing on 4 nodes

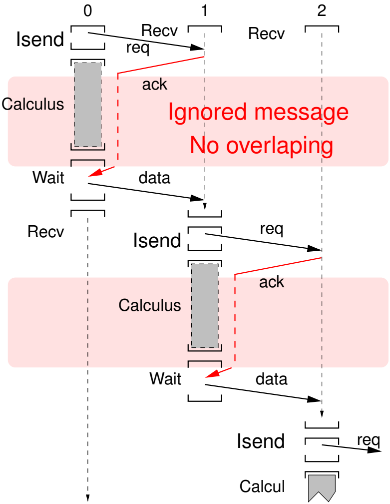
```
if (mynode!=0)
  MPI_Recv();

req=MPI_Isend(next);
Work(); /* about 1s */
MPI_Wait(req);

if (mynode==0)
  MPI_Recv();
```

- expected time: ~ 1 s
- observed time: ~ 4 s

Asynchronous communications with MPI



Token circulation while computing on 4 nodes

```
if (mynode!=0)
  MPI_Recv();

req=MPI_Isend(next);
Work(); /* about 1s */
MPI_Wait(req);

if (mynode==0)
  MPI_Recv();
```

- expected time: ~ 1 s
- observed time: ~ 4 s

Asynchronous communications

Problems: asynchronous communications required

- progression of asynchronous communications (MPI)
- remote PUT/GET primitives
- etc.

Solutions

- Using threads
- Implementing part of the protocol in the network card (MPICH/GM)
- Using remote memory reads

Multithreading

A solution for asynchronous communications

- computations can overlap communications
- automatic parallelism

But disparity of implementations

- kernel threads
  - blocking system calls, SMP
- users threads
  - efficient, flexible
- mixed model threads

Difficulties of threads and communications

Different way to communicate

- active polling
  - memory read, non blocking system calls
- passive polling
  - blocking system calls, signals

Different usable methods

- not always available
- not always compatible
  - with the operating system
  - with the application

An experimental proposition: an I/O server

Requests centralization

- a service for the application
- allow optimizations
  - aggregation of requests

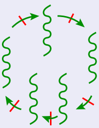
Portability of the application

- uniform interface
  - effective strategies (polling, signals, system calls) are hidden to the application
- application without explicit strategy
  - independence from the execution plate-form

I/O server linked to the thread scheduler

Threads and polling

- difficult to implement
- the thread scheduler can help to get guarantee frequency for polling
  - independent with respect to the number of threads in the application



instead of

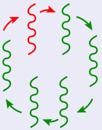


Illustration of such an interface

Registration of events kinds

`IO_handle=IO_register(params)`

- call-back functions registration
- used by communication libraries at initialization time

Waiting for an event

`IO_wait(IO_handle, arg)`

- blocking function for the current thread
- the scheduler will use the call-backs
  - communications are still manged by communication libraries

Example with MPI

Registration

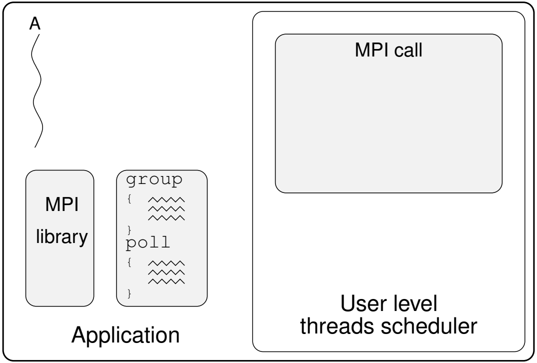
```
IO_t MPI_IO;
...
IO_register_t params = {
 .blocking_syscall:=NULL,
 .group=&group_MPI(),
 .poll=&poll_MPI(),
 .frequency=1
};

MPI_IO=
 IO_register(&params);
...
```

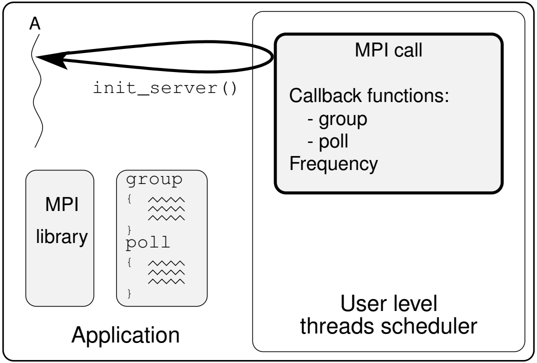
Communication

```
MPI_Request request;
IO_MPI_param_t param;
...
MPI_Irecv(buf, size,
    ..., &request);
param.request=&request;
IO_wait(MPI_IO, &param);
...
```

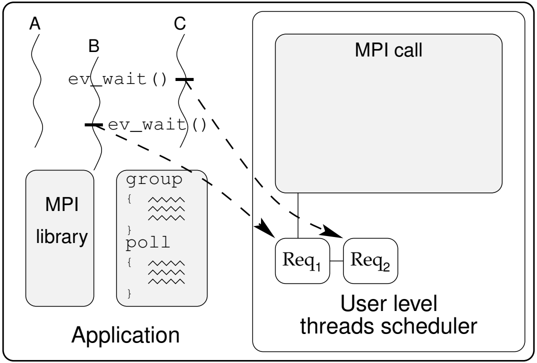
Running the scrutation server



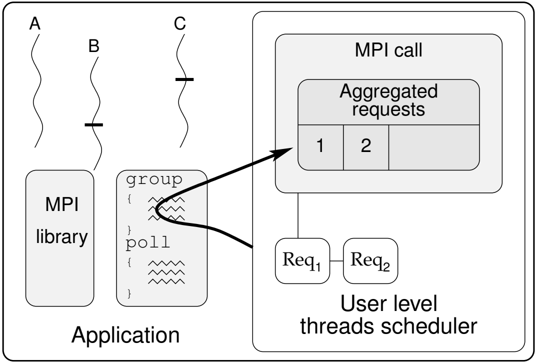
Running the scrutation server



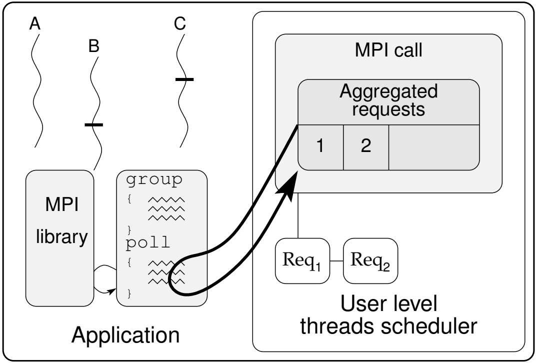
Running the scrutation server



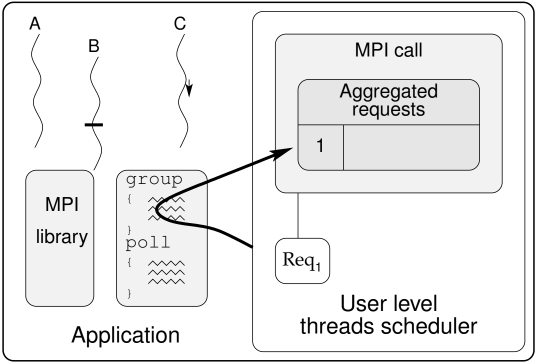
Running the scrutation server



Running the scrutation server



Running the scrutation server



Key points

High level communication libraries needs multithreading

- allow independent communication progression
- allow asynchronous operations (puts/gets)

Threads libraries must be designed with services for communication libraries

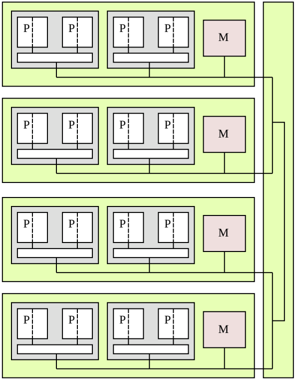
- allow efficient polling
- allow selection of communication strategy

Outlines

Towards more and more hierarchical computers

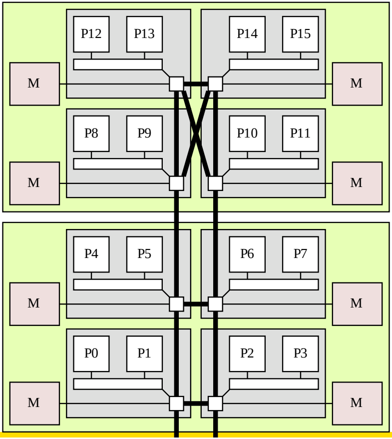
- SMT

  (HyperThreading)
- Multi-Core
- SMP
- Non-Uniform Memory Access (NUMA)



Hagrid, octo-dual-core

- AMD Opteron
- NUMA factor 1.1-1.5

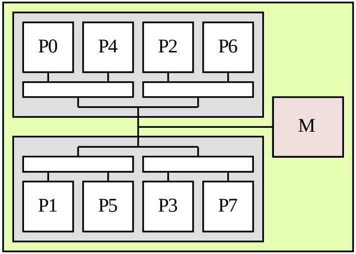


11

Courtesy of Samuel Thibault

Aragog, dual-quad-core

- Intel
- Hierarchical cache levels



Courtesy of Samuel Thibault

How to run applications on such machines?

Courtesy of Samuel Thibault

How to program parallel machines?

- By hand
  - Tasks, POSIX threads, explicit context switch
- High-level languages
  - Processes, task description, OpenMP, HPF, UPC, ...

- Technically speaking, threads

- How to schedule them efficiently?

Courtesy of Samuel Thibault

How to schedule efficiently?

- Performance
  - Affinities between threads and memory taken into account
- Flexibility
  - Execution easily guided by applications
- Portability
  - Applications adapted to any new machine

Courtesy of Samuel Thibault

Predetermined approaches

- Two phases
  - Preliminary computation of
    - Data placement [Marather, Mueller, 06]
    - Thread scheduling
  - Execution
    - Strictly follows the pre-computation
- Example: PaStiX [Hénon, Ramet, Roman, 00]

✔ Excellent performances

✗ Not always sufficient or possible: strongly irregular problems...

Courtesy of Samuel Thibault

Opportunistic approaches

- Various greedy algorithms
  - Single / several [Markatos, Leblanc, 94] / a hierarchy of task lists [Wang, Wang, Chang, 00]
- Used in nowaday's operating systems
  - Linux, BSD, Solaris, Windows, ...

✔ Good portability

✗ Uneven performances
  - No affinity information...

Negotiated approaches

- Language extensions
  - OpenMP, HPF, UPC, ...

✔ Portability (adapts itself to the machine)

✗ Limited expressivity (e.g. no NUMA support)

- Operating System extensions
  - NSG, liblgroup, libnuma, ...

✔ Freedom for programmers

✗ Static placement, requires rewriting placement strategies according to the architecture

Courtesy of Samuel Thibault

Issues

- Negotiated approach seems promising, but
  - Which scheduling strategy?
    - Depends on the application
  - Which information to take into account?
    - Affinities between threads?
    - Memory occupation?
  - Where does the runtime play a role?
- But there is hope!
  - Programmers and compilers do have some clues to give
  - Missing piece: structures

Courtesy of Samuel Thibault

BubbleSched

Guiding scheduling through bubbles

Courtesy of Samuel Thibault

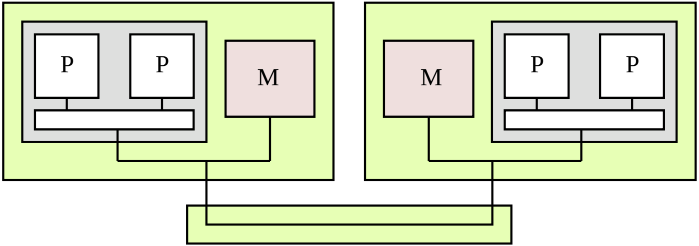
Idea: Structure to better schedule

Bridging the gap between programmers and architectures

- Grab the structure of the parallelism
  - Express relations between threads, memory, I/O, ...
- Model the architecture in a generic way
  - Express the structure of the computation power
- Scheduling is mapping
  - As it should just be!
  - Completely algorithmic
  - Allows all kinds of scheduling approaches

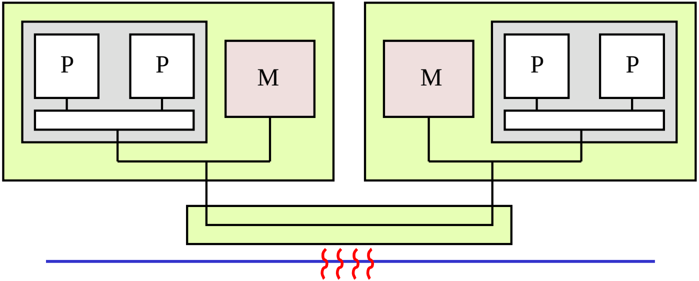
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



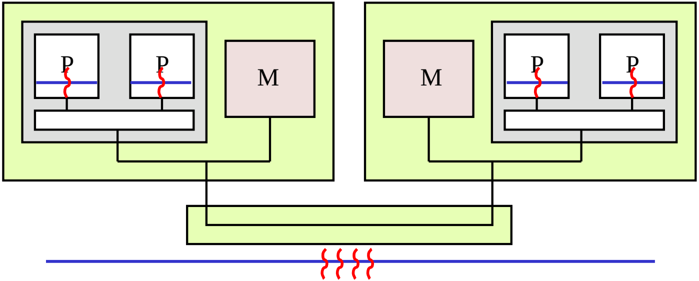
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



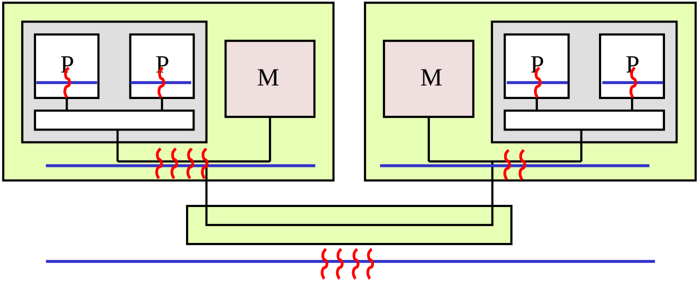
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



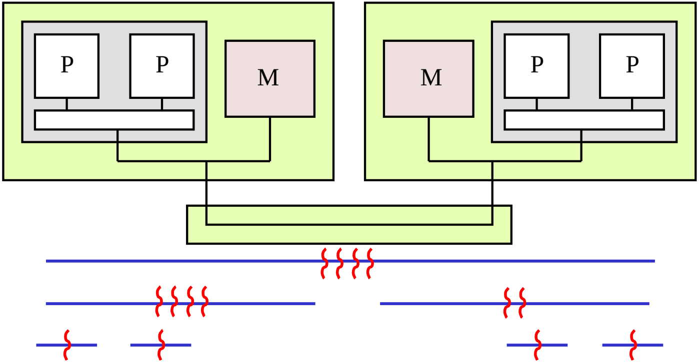
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



Courtesy of Samuel Thibault

Runqueues to model hierarchical machines

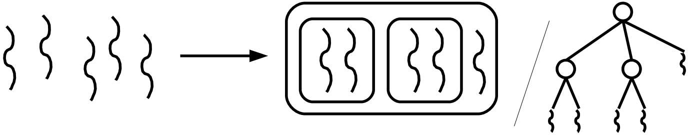


Courtesy of Samuel Thibault

Bubbles to model thread affinities

Keeping the structure of the application in mind

- Data sharing
- Collective operations
- ...



```
bubble_insert_thread(bubble, thread);
bubble_insert_bubble(bubble, subbubble);
```

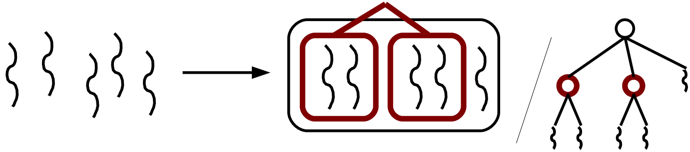

Courtesy of Samuel Thibault

Bubbles to model thread affinities

Keeping the structure of the application in mind

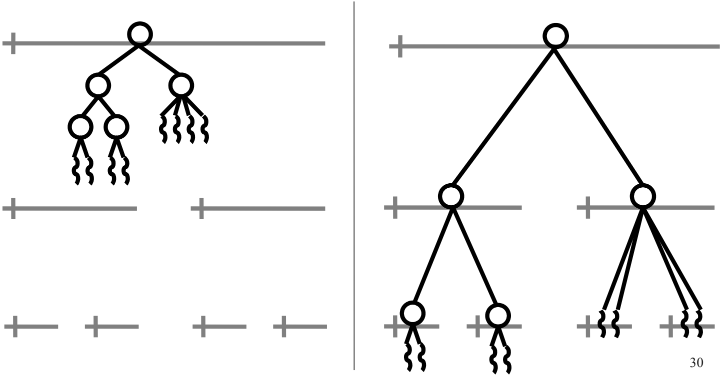
- Data sharing
- Collective operations
- ...

Some can be stronger



bubble_insert_thread(bubble, thread);
bubble_insert_bubble(bubble, subbubble);

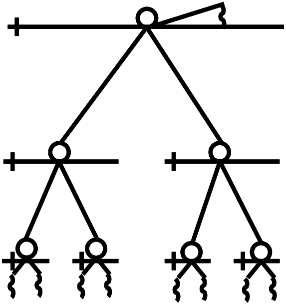
Examples of thread and bubble repartitions



Courtesy of Samuel Thibault

Implemented schedulers

- Full-featured schedulers
  - Gang scheduling
  - Spread
    - Favor load balancing
  - Affinity
    - Favor affinities (Broquedis)
    - Memory aware (Jeuland)
- Reuse and compose
  - Work stealing
  - Combined schedulers (time, space, etc.)



Courtesy of Samuel Thibault

Conclusion
A new scheduling approach

Structure & conquer!

- Bubbles = simple yet powerful abstractions
  - Recursive decomposition schemes
    - Divide & Conquer
    - OpenMP
- Implement scheduling strategies for hierarchical machines
  - A lot of technical work is saved
- Significant benefits
  - 20-40%

Courtesy of Samuel Thibault

Outlines

High-performance parallel programming is difficult

Need of efficiency

- lots of efficient hardware available (network, processors, etc.)
- but lots of API

Need of portability

- applications cannot be rewritten for each new hardware
- use of standard interfaces (pthread, MPI, etc.)

On the way to the portability of the efficiency

- very difficult to get: still lots of research
- require very well designed interfaces allowing:
  - the application to describe its behavior (semantics)
  - the middle-ware to select the strategies
  - the middle-ware to optimize the strategies

Three examples from research projects

- Madeleine: an efficient and portable communication library
  - optimization of communication strategies
- Marcel: an I/O server in a thread scheduler
  - efficient management of threads with communications
- BubbleSched: a scheduler for hierarchical plate-forms
  - efficient scheduling on hierarchical machines

Three efficient middlewares for specific aspects

- lots of criteria to optimize in real applications
  - scheduling, communication, memory, etc.
- multi-criteria optimization is more than aggregation of mono-criteria optimization
- other high-level interface programming for parallel applications ? (work-stealing, etc.)