HPC 101 (cont.)

Arnaud LEGRAND, CR CNRS, LIG/INRIA/Mescal

Vincent DANJEAN, MCF UJF, LIG/INRIA/Moais

November, 12th 2012

Goals of the two next lectures

Learn and understand low-level software in HPC

Understand the internal of HPC programming model implementations

Limitation of mixing HPC programming models

Low-level API in HPC (cont.)

Mixing HPC libraries

Programming for HPC

Part IV

Low-level API in HPC (cont.)

Outlines: Low-level API in HPC (cont.)

2. OpenMP
   - Presentation
   - Examples
3. OpenCL and Cuda
4. An API is not an implementation

What is OpenMP?

An API to parallelize a program

explicitly, with threads, with shared memory

Contents of OpenMP

- compiler directives
- runtime library routines
- environment variables

OpenMP abbreviation

Short version Open Multi-Processing

Long version Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

What is not OpenMP?

- not designed to manage distributed memory parallel systems
- implementation can vary depending on the vendor
- no optimal performance guarantee
- not a checker for data dependencies, deadlock, etc.
- not a checker for code correction
- not a automatic parallelization tool
- not designed for parallel I/O

More information

https://computing.llnl.gov/tutorials/openMP/
http://openmp.org/wp/

Goals of OpenMP

Standardization

- target a variety of shared memory architectures/platforms
- supported by lots of hardware and software vendors

Lean and Mean (less pertinent with last releases)

- simple and limited set of directives
- 3 or 4 directives enough for classical parallel programs

Ease of Use

- allows to incrementally parallelize a serial program
- allows both coarse-grain and fine-grain parallelism

Portability (API in C/C++ and Fortran)

- public forum for API and membership
- most major platforms have been implemented

C/C++ general code structure

```
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code
  // Beginning of parallel section. Fork a team of threa
  // Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
  {
    Parallel section executed by all threads
    Other OpenMP directives
    Run-time Library calls
    All threads join master thread and disband
  }
  Resume serial code
}
```

C/C++ `for` Directive Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
main() {
  int i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
# pragma omp parallel shared(a,b,c,chunk) private(i)
  {
#   pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

Outlines: Low-level API in HPC (cont.)

Parallel Programming with GPU

GPGPU: General Purpose Graphic Processing Unit

- very good ratio GFlops/price and GFlops/Watt
- GPU Tesla C2050 from NVidia : about 300 GFlops in double precision
- specialized hardware architecture:
  classical programming does not work

Two leading environments

Cuda specific to NVidia, can use all the features of NVidia cards. Works only with NVidia GPU.

OpenCL norm (not implementation) supported by different vendors (AMD, NVidia, Intel, Apple, etc.) Target GPUs but also CPUs.

Very similar programming concepts

Cuda and OpenCL bases

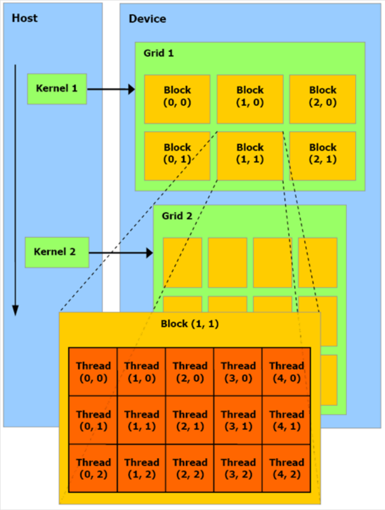
Part 1: device programs

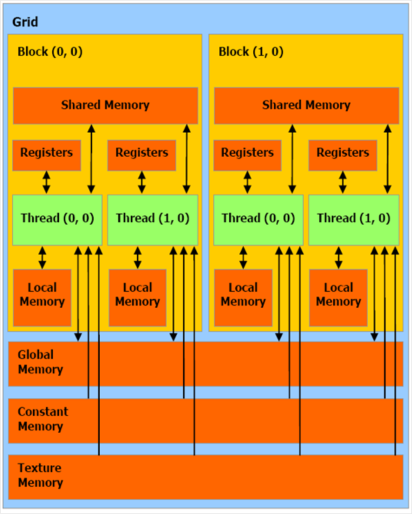
- C code with restriction and extension (memory model, vector types, etc.)
- run in parallel by lots of threads on the targeted hardware
- functions to be run are called kernels

Part 2: host programs

- API in C/C++
- manage memory transfers
- manage kernel launches (compilations and runs)

CUDA architecture model





OpenCL device program

Classical program

```
void vector_add_cpu (const float* src_a,
  const float* src_b, float* res, const int num)
{
   for (int i = 0; i < num; i++)
      res[i] = src_a[i] + src_b[i];
}
```

OpenCL program

```
__kernel void vector_add_gpu (
              __global const float* src_a,
              __global const float* src_b,
              __global float* res, const int num) {
   const int idx = get_global_id(0);
   if (idx < num)
      res[idx] = src_a[idx] + src_b[idx];
}
```

Outlines: Low-level API in HPC (cont.)

2. OpenMP
3. OpenCL and Cuda
4. An API is not an implementation
    - Linux POSIX Threads Libraries

Linux POSIX Threads Libraries: history

LinuxThread (1996) : kernel level, Linux standard thread library for a long time, not fully POSIX compliant

GNU-Pth (1999) : user level, portable, POSIX

NGPT (2002) : mixed, based on GNU-Pth, POSIX, not developed anymore

NPTL (2002) : kernel level, POSIX, current Linux standard thread library

PM2/Marcel (2001) : mixed, POSIX compliant, lots of extensions for HPC (scheduling control, etc.)

Mutex, etc. implementation

From signals...

- communication base of the linuxthread library
- the only support from the kernel at this time
- one 'manager' hidden thread
- race conditions and error prone
- not really efficient

...to futex

- synchronization in userspace (no system call) if no contention
- allow synchronization between processes
- specific support from the kernel
- used in nptl

Part V

Mixing HPC libraries

Outlines: Mixing HPC libraries

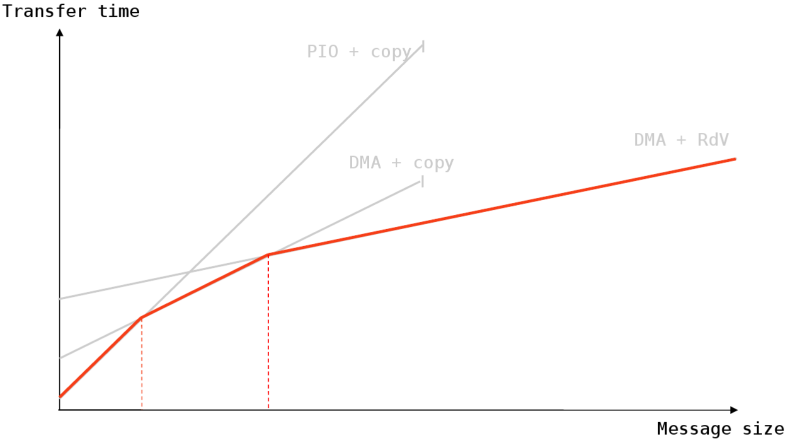
5. Optimizing communications
   - Optimizing communication methods
   - An experimental project: the Madeleine interface
6. Asynchronous communications
7. Hierarchical plate-forms and efficient scheduling
8. Conclusion

Optimizing communication methods

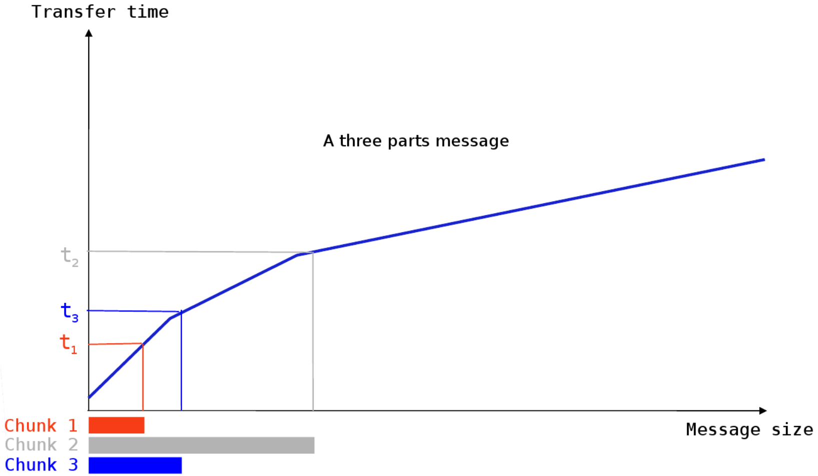
Low-level libraries sometimes prefer using the processor in order to guaranty low latencies

- 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

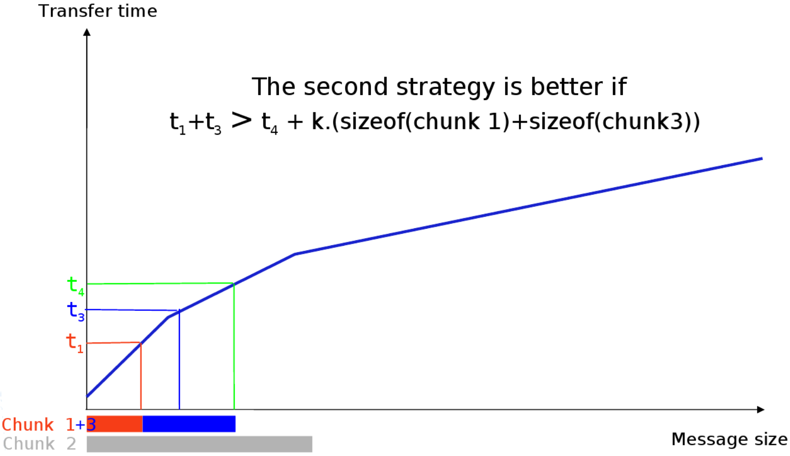
Choosing the Optimal Strategy



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 ?

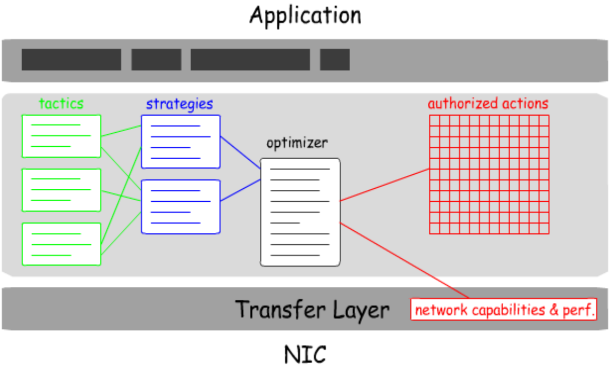
Using parameters and historic

- 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: Mixing HPC libraries

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

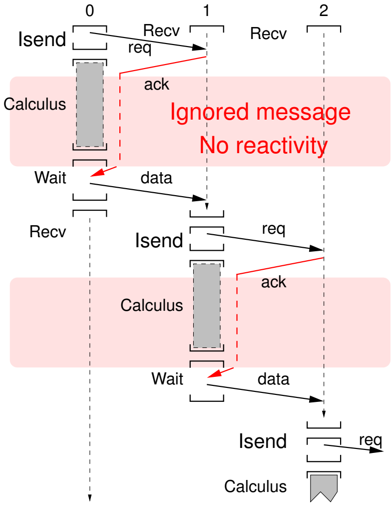
```
if (mynode!=0)
  MPI_Recv();

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

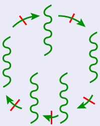
Portability of the application

- 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

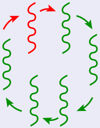


Illustration of such an interface

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

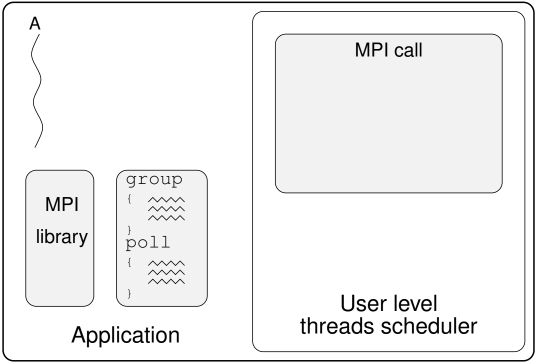
```
IO_t MPI_IO;
...
IO_register_t params = {
 .blocking_syscall:=NULL,
 .group=&group_MPI(),
 .poll=&poll_MPI(),
 .frequency=1
};

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

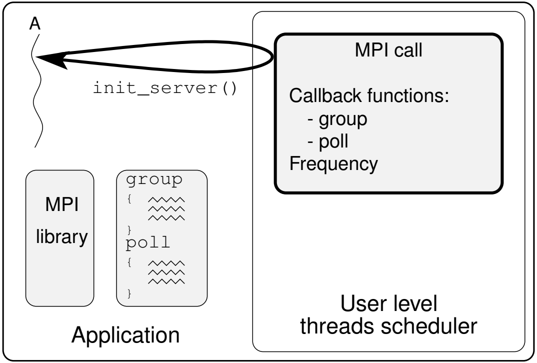
Communication

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

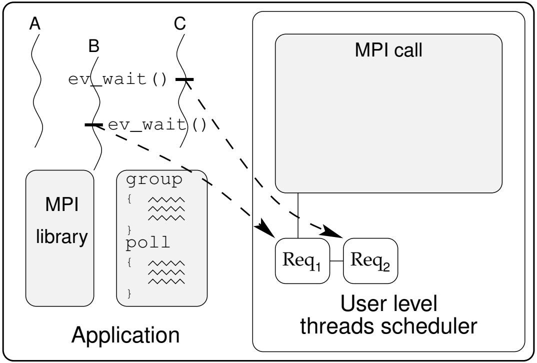
Running the scrutation server



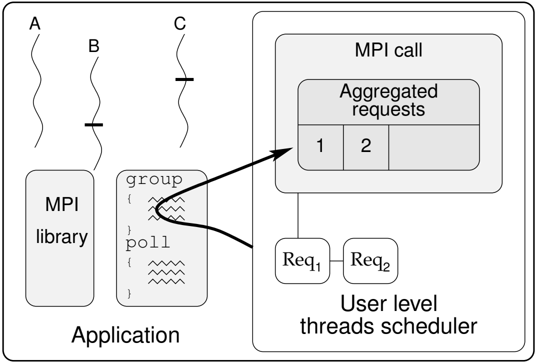
Running the scrutation server



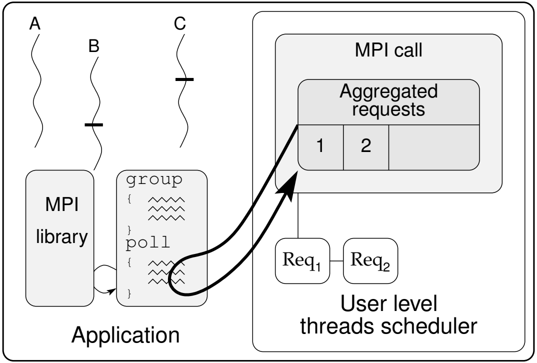
Running the scrutation server



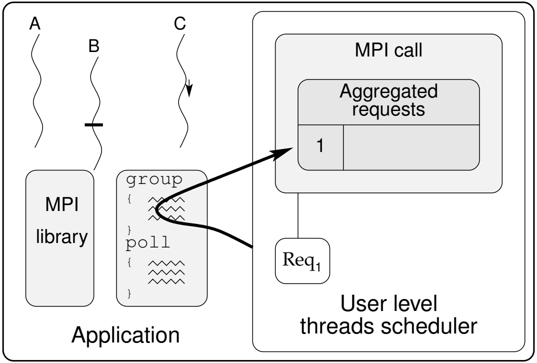
Running the scrutation server



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)

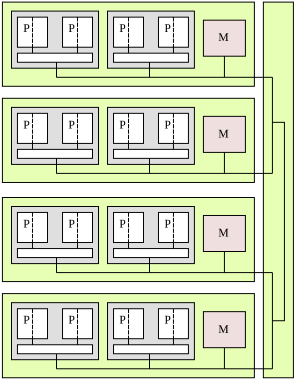
Threads libraries must be designed with services for communication libraries

- allow efficient polling
- allow selection of communication strategy

Outlines: Mixing HPC libraries

Towards more and more hierarchical computers

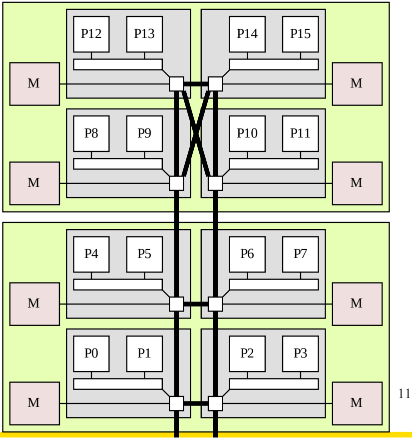
- SMT
  (HyperThreading)
- Multi-Core
- SMP
- Non-Uniform Memory Access (NUMA)



Courtesy of Samuel Thibault

Hagrid, octo-dual-core

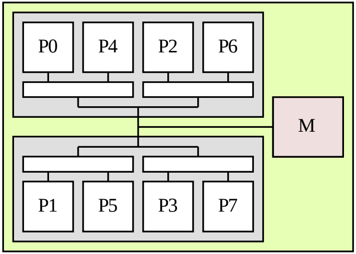
- AMD Opteron
- NUMA factor 1.1-1.5



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...

Courtesy of Samuel Thibault

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

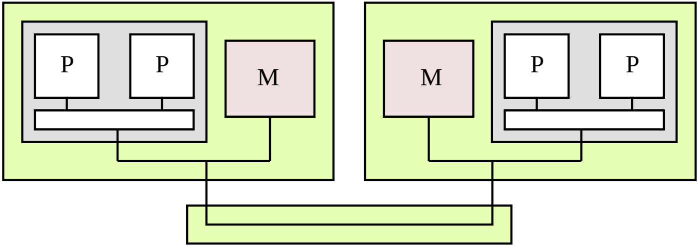
Idea: Structure to better schedule

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

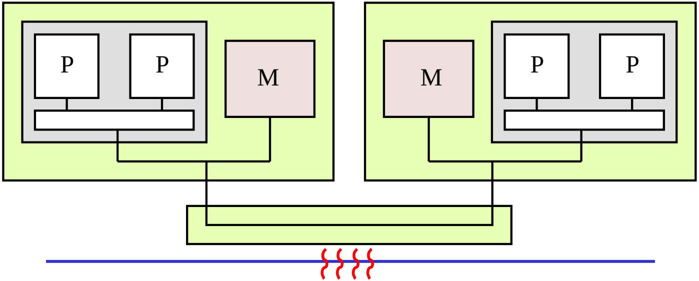
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



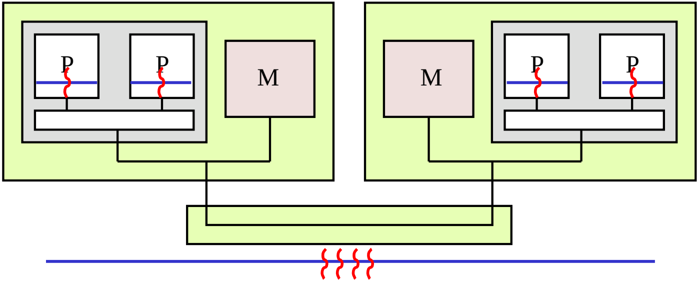
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



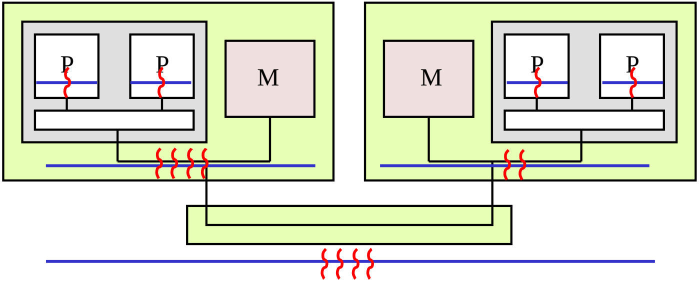
Courtesy of Samuel Thibault

Runqueues to model hierarchical machines

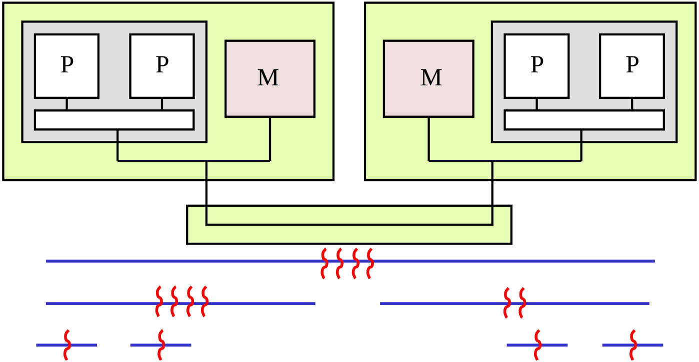


Courtesy of Samuel Thibault

Runqueues to model hierarchical machines



Runqueues to model hierarchical machines

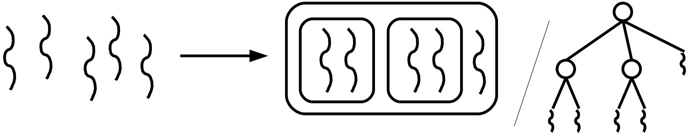


Courtesy of Samuel Thibault

Bubbles to model thread affinities

Keeping the structure of the application in mind

- Data sharing
- Collective operations
- ...

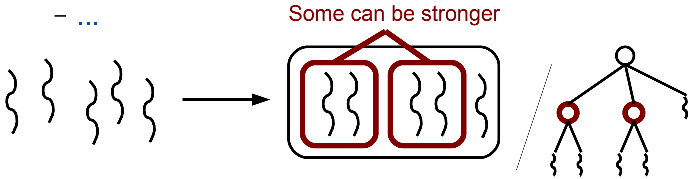


```
bubble_insert_thread(bubble, thread);
bubble_insert_bubble(bubble, subbubble);
```

Bubbles to model thread affinities

Keeping the structure of the application in mind

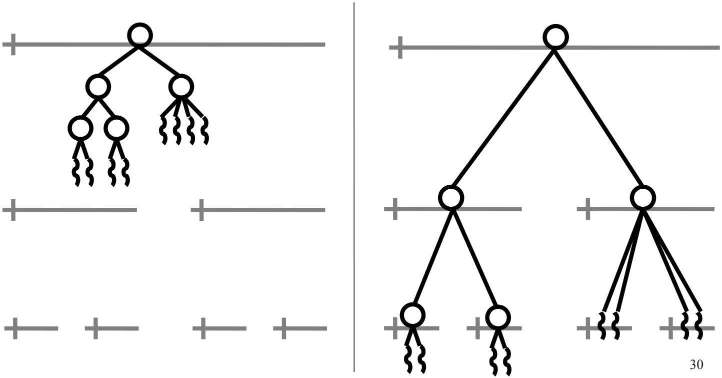
- Data sharing
- Collective operations
- ...



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

Courtesy of Samuel Thibault

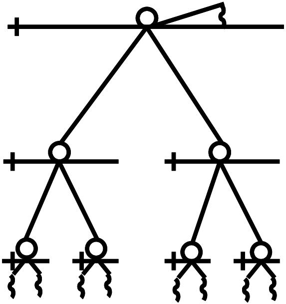
Examples of thread and bubble repartitions



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%

Outlines: Mixing HPC libraries

5 Optimizing communications

6 Asynchronous communications

7 Hierarchical plate-forms and efficient scheduling

8 Conclusion
- High-performance parallel programming is difficult

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.)

Part VI

Programming for HPC

Matrix multiplication in MPI

How to write such a program?