How to Efficiently Program High Performance Architectures ?

Arnaud LEGRAND, CR CNRS, LIG/INRIA/Mescal
Jean-Louis ROCH, MCF ENSIMAG, LIG/INRIA/Moais

Vincent DANJEAN, MCF UJF, LIG/INRIA/Moais
Derick KONDO, CR INRIA, LIG/INRIA/Mescal
Jean-François MÉHAUT, PR UJF, LIG/INRIA/Mescal
Bruno RAFFIN, CR INRIA, LIG/INRIA/Moais
Alexandre TERMIER, MCF UJF, LIG/Hadas

Some slides come from Samuel THIBAULT, MCF Bordeaux, LaBRI/Runtime

October, 6th 2008

High Performance Computing

Needs are always here

- numerical or financial simulation, modelisation, virtual reality virtuelle
- more data, more details, . . .

Computing power will never be enough

One way to follow: using parallelism

Idea: change space into time

more resources to gain some time

High Performance Architectures

Parallelism and Threads

Synchronisation

Multithreading and Networking

12 A Brief Overview of MPI

13 Mixing Threads and Communication in HPC
- Problems arises
- Discussion about solution

Part I

High Performance Architectures

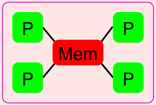
Outlines: High Performance Architectures

Parallel Architectures

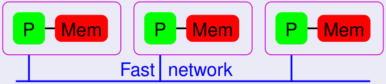
Two main kinds

Architectures with shared memory and architectures with distributed memory.

Multiprocessors



Clusters



Why several processors/cores ?

Limits for monocore processors

- superscalar processors: instruction level parallelism
- frequency
- electrical power

What to do with place available on chips ?

- caches (bigger and quicker)
- several series of registers (hyperthreaded processors)
- several series of cores (multi-core processors)
- all of that

Symmetric Multi Processors

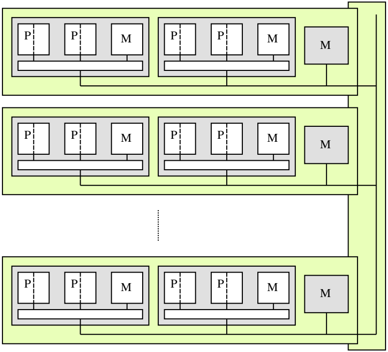
- all processors have access to the same memory and I/O
- most common multiprocessor systems today use an SMP architecture
- in case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors

Non Uniform Memory Access Architectures

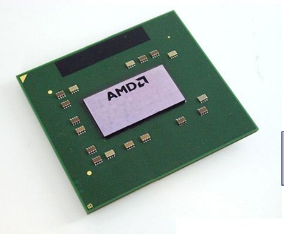
- memory access time depends on the memory location relative to a processor
- better scaling hardware architecture
- harder to program efficiently: trade off needed between load-balancing and memory data locality

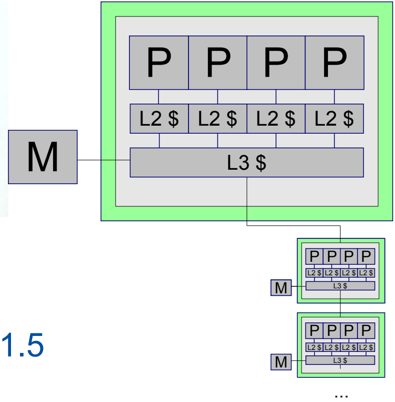
Towards more and more hierarchical computers

- SMT (HyperThreading)
- Multi-core
- NUMA



AMD Quad-Core

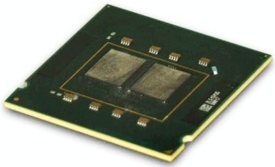


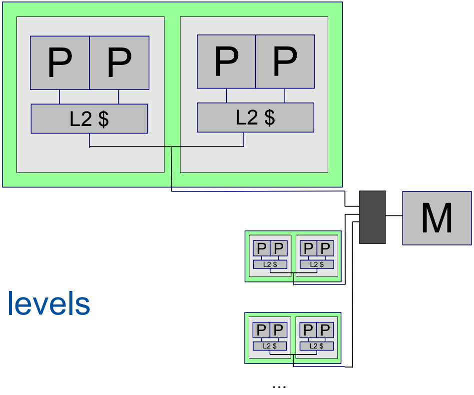


Shared L3 cache

NUMA factor ~1.1-1.5

Intel Quad-Core

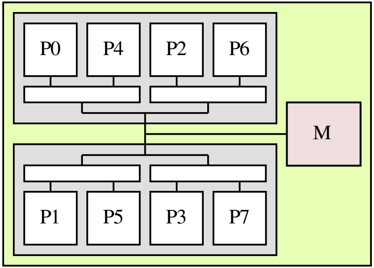




Hierarchical cache levels

dual-quad-core

- Intel
- Hierarchical cache levels



Clusters

Composed of a few to hundreds of machines

- often homogeneous
  - same processor, memory, etc.
- often linked with a high speed, low latency network
  - Myrinet, InfinityBand, Quadrix, etc.

Biggest clusters can be split in several parts

- computing nodes
- I/O nodes
- front (interactive) node

Grids

Lots of heterogeneous resources

- aggregation of clusters and/or standalone nodes
- high latency network (Internet for example)
- often dynamic resources (clusters/nodes appear and disappear)
- different architectures, networks, etc.

Current Architectures in HPC

Hierarchical Architectures

- HT technology
- multi-core processor
- multi processors machine
- cluster of machines
- grid of clusters and individual machines

Even more complexity

- computing on GPU
  - require specialized codes but hardware far more powerful
- FPGA
  - hardware can be specialized on demand
  - still lots of work on interface programming here

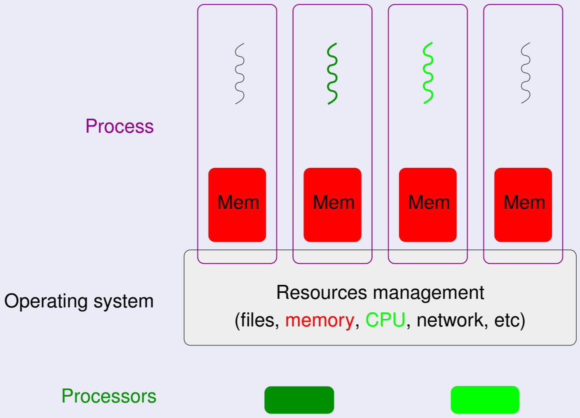
Part II

Parallelism and Threads

Outlines: Parallelism and Threads

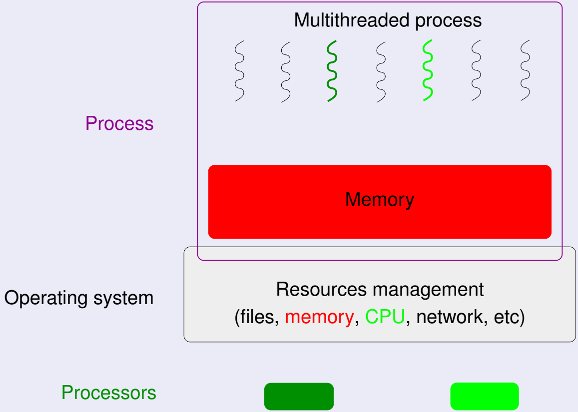
Programming on Shared Memory Parallel Machines

Using process



Programming on Shared Memory Parallel Machines

Using threads



Introduction to Threads

Why threads ?

- To take profit from shared memory parallel architectures
  - SMP, hyperthreaded, multi-core, NUMA, etc. processors

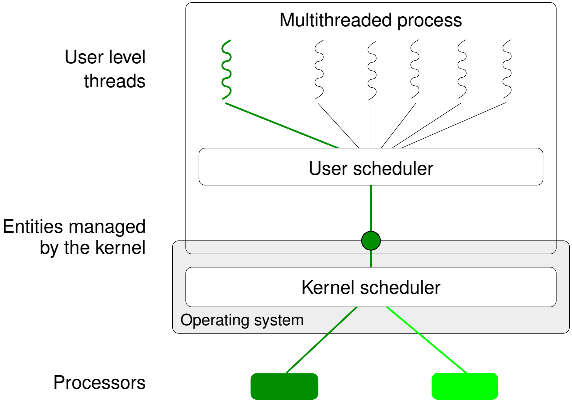
    future Intel processors: several hundreds cores
- To describe the parallelism within the applications
  - independent tasks, I/O overlap, etc.

What will use threads ?

- User application codes
  - directly (with thread libraries)

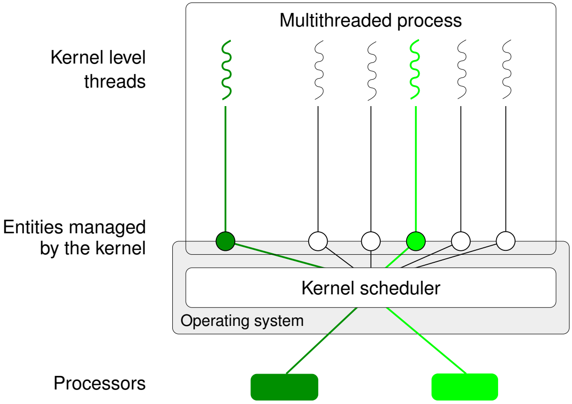
    POSIX API (IEEE POSIX 1003.1C norm) in C, C++, . . .
  - with high-level programming languages (Ada, OpenMP, . . .)
- Middleware programming environments
  - demonized tasks (garbage collector, . . .), . . .

User threads



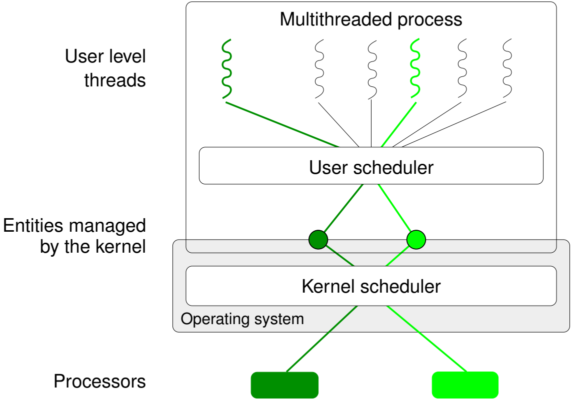
Efficiency + Flexibility + SMP - Blocking syscalls -

Kernel threads



Efficiency - Flexibility - SMP + Blocking syscalls +

Mixed models



Efficiency + Flexibility + SMP + Blocking syscalls limited

Mixed models

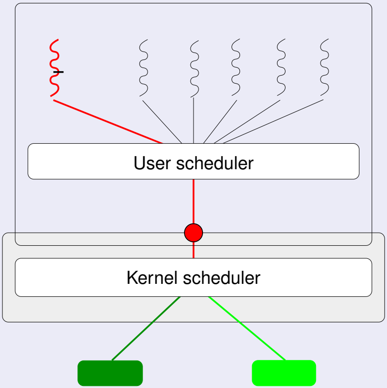
	Characteristics			
Library	Efficiency	Flexibility	SMP	Blocking syscalls
User	+	+	-	-
Kernel	-	-	+	+
Mixed	+	+	+	limited

Summary

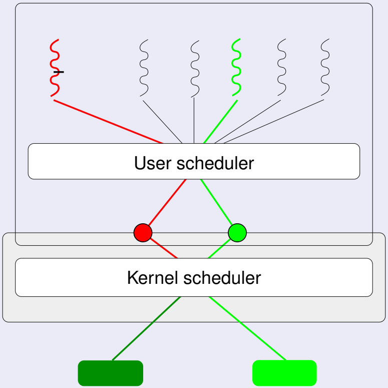
Mixed libraries seems more attractive however they are more complex to develop. They also suffer from the blocking system call problem.

User Threads and Blocking System Calls

User level library



Mixed library



Scheduler Activations

Idea proposed by Anderson et al. (91)

Dialogue (and not monologue) between the user and kernel schedulers

- the user scheduler uses system calls
- the kernel scheduler uses upcalls

Upcalls

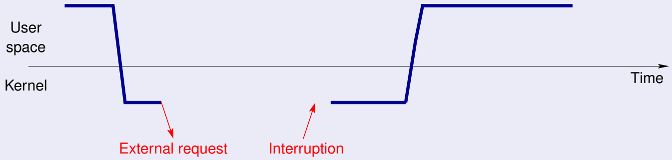
Notify the application of scheduling kernel events

Activations

- a new structure to support upcalls
  a kinf of kernel thread or virtual processor
- creating and destruction managed by the kernel

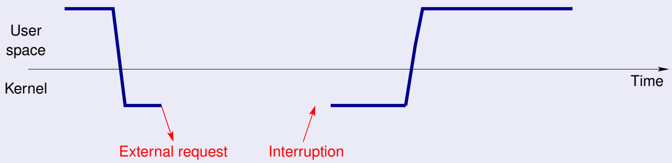
Scheduler Activations

Instead of:

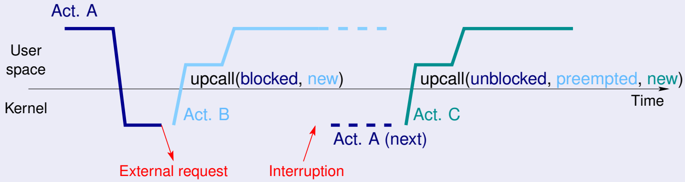


Scheduler Activations

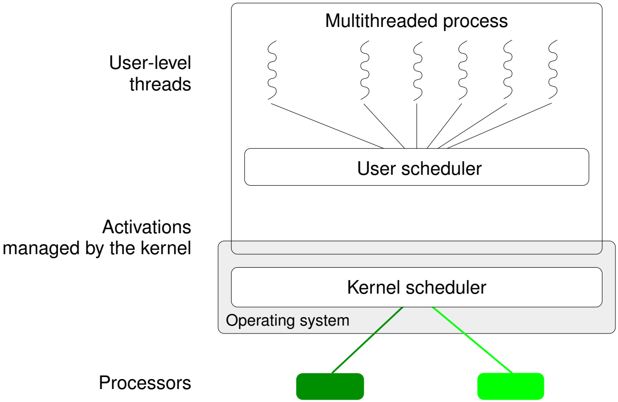
Instead of:



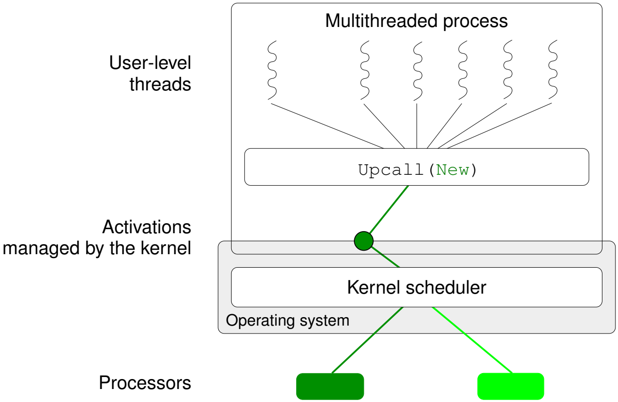
...better use the following schema:



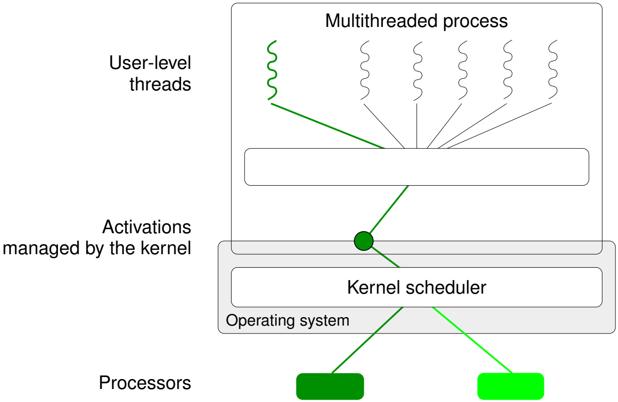
Working principle



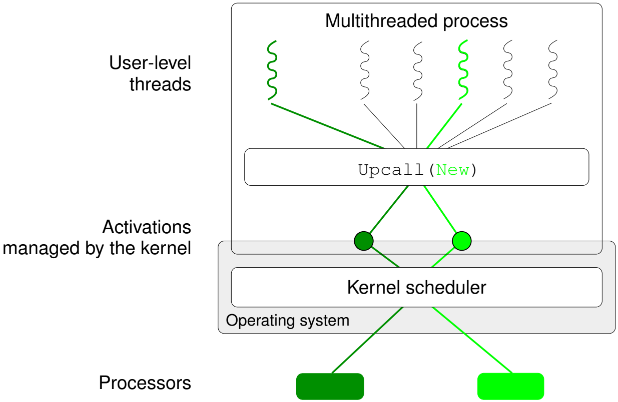
Working principle



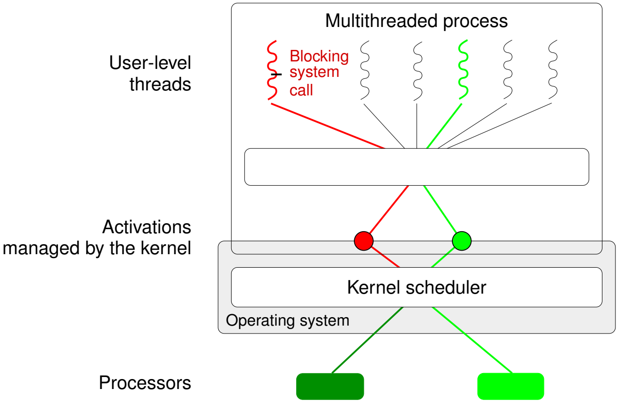
Working principle



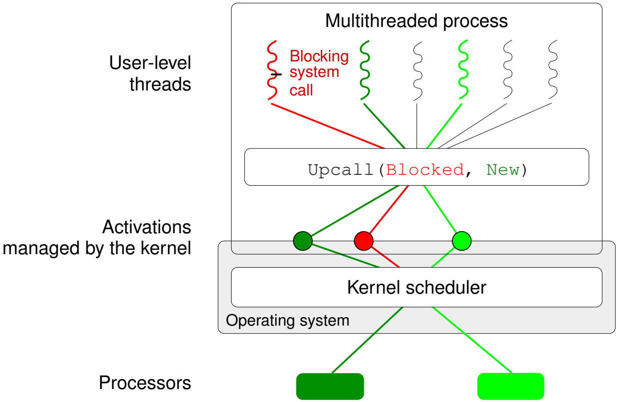
Working principle



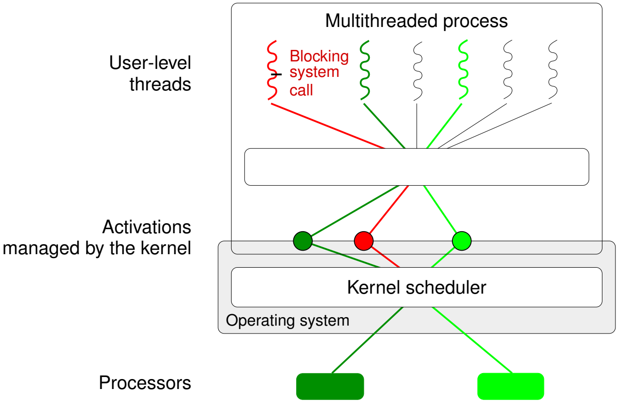
Working principle



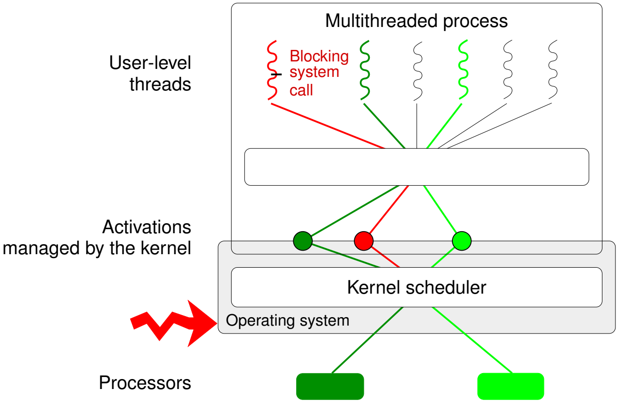
Working principle



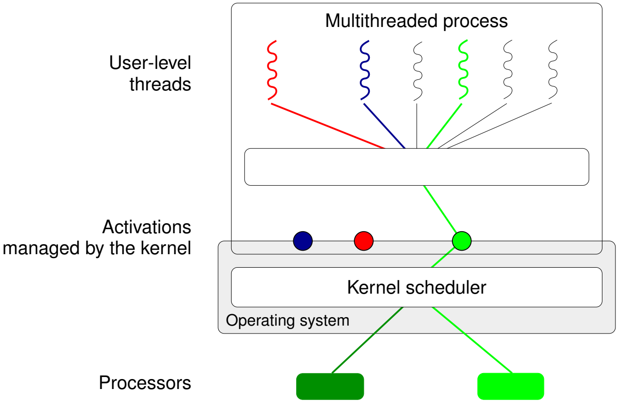
Working principle



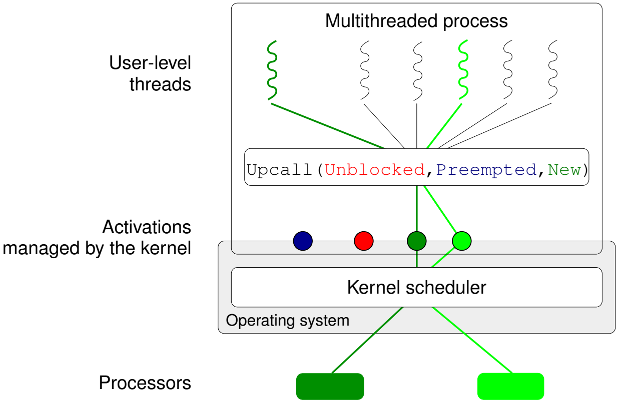
Working principle



Working principle



Working principle



Normalisation of the thread interface

Before the norm

- each Unix had its (slightly) incompatible interface
- but same kinds of features was present

POSIX normalisation

- IEEE POSIX 1003.1C norm (also called POSIX threads norm)
- Only the API is normalised (not the ABI)
  - POSIX thread libraries can easily be switched at source level but not at runtime
- POSIX threads own
  - processor registers, stack, etc.
  - signal mask
- POSIX threads can be of any kind (user, kernel, etc.)

Linux POSIX Threads Libraries

- **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.)

Basic POSIX Thread API

Creation/destruction

- `int` **`pthread_create`**`(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)`
- `void` **`pthread_exit`**`(void *value_ptr)`
- `int` **`pthread_join`**`(pthread_t thread, void **value_ptr)`

Synchronisation (semaphores)

- `int` **`sem_init`**`(sem_t *sem, int pshared, unsigned int value)`
- `int` **`sem_wait`**`(sem_t *sem)`
- `int` **`sem_post`**`(sem_t *sem)`
- `int` **`sem_destroy`**`(sem_t *sem)`

Basic POSIX Thread API (2)

Synchronisation (mutex)

- `int` **`pthread_mutex_init`**`(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
- `int` **`pthread_mutex_lock`**`(pthread_mutex_t *mutex)`
- `int` **`pthread_mutex_unlock`**`(pthread_mutex_t *mutex)`
- `int` **`pthread_mutex_destroy`**`(pthread_mutex_t *mutex)`

Synchronisation (conditions)

- `int` **`pthread_cond_init`**`(pthread_cond_t *cond, const pthread_condattr_t *attr)`
- `int` **`pthread_cond_wait`**`(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `int` **`pthread_cond_signal`**`(pthread_cond_t *cond)`

Basic POSIX Thread API (3)

Per thread data

- `int` **`pthread_key_create`**`(pthread_key_t *key, void (*destr_function) (void*))`
- `int` **`pthread_key_delete`**`(pthread_key_t key)`
- `int` **`pthread_setspecific`**`(pthread_key_t key, const void *pointer)`
- `void *` **`pthread_getspecific`**`(pthread_key_t key)`

Basic POSIX Thread API (3)

Per thread data

- `int` **`pthread_key_create`**`(pthread_key_t *key, void (*destr_function) (void*))`
- `int` **`pthread_key_delete`**`(pthread_key_t key)`
- `int` **`pthread_setspecific`**`(pthread_key_t key, const void *pointer)`
- `void *` **`pthread_getspecific`**`(pthread_key_t key)`

The new `__thread` C keyword

- used for a global per-thread variable
- need support from the compiler and the linker at compile time and execute time
- libraries can have efficient per-thread variables without disturbing the application

Part III

Synchronisation

Outlines: Synchronisation

Hardware Support

What happens with incrementations in parallel?

`var++;`	`var++;`

Hardware Support

What happens with incrementations in parallel?

```
for (i=0; i<10; i++){
  var++;
}
```

```
for (i=0; i<10; i++){
  var++;
}
```

Hardware Support

What happens with incrementations in parallel?

`for (i=0; i<10; i++){`
`  var++;`
`}`
	`for (i=0; i<10; i++){`
	`  var++;`
	`}`

Hardware support required

TAS atomic test and set instruction

cmpexchge compare and exchange

atomic operation incrementation, decrementation, adding, etc.

Critical section with busy waiting

Example of code

```
while (!  TAS(&var))
      ;
/* in critical section */
var=0;
```

Critical section with busy waiting

Example of code

```
while (!  TAS(&var))
       while (var) ;
/* in critical section */
var=0;
```

Critical section with busy waiting

Example of code

```
while (!  TAS(&var))
       while (var) ;
/* in critical section */
var=0;
```

Busy waiting

- \+ very reactive
- \+ no OS or lib support required
- \- use a processor while not doing anything
- \- does not scale if there are lots of waiters

Semaphores

- Internal state: a counter initialised to a positive or null value
- Two methods:
  - P(s) wait for a positive counter then decrease it once
  - V(s) increase the counter

Common analogy: a box with tokens

- Initial state: the box has *n* tokens in it
- One can put one more token in the box (V)
- One can take one token from the box (P) waiting if none is available

Monitors

Mutex

- Two states: locked or not
- Two methods:

  lock(m) take the mutex

  unlock(m) release the mutex (must be done by the thread owning the mutex)

Conditions

- waiting thread list (conditions are not related with tests)
- Three methods:

  wait(c, m) sleep on the condition. The mutex is released atomically during the wait.

  signal(c) one sleeping thread is wake up

  broadcast(c) all sleeping threads are wake up

Old Linux libpthread

First Linux kernel thread library

- limited kernel support available
- provides POSIX primitives (mutexes, conditions, semaphores, etc.)

All internal synchronisation built on signals

- lots of play with signal masks
- one special (manager) thread used internally to manage thread state and synchronisation
- race conditions not always handled (not enough kernel support)

NPTL: New POSIX Thread Library

New Linux kernel thread library

- requires new kernel support (available from Linux 2.6)
- specific support in the libc
- a lot more efficient
- fully POSIX compliant

Internal synchronisation based on futex

- new kernel object
- mutex/condition/semaphore can be fully handled in user space unless there is contention

Part IV

Multithreading and Networking

Outlines: Multithreading and Networking

Standard in industry

- frequently used by engineers, physicians, etc.
- MPI2 begin to be available

See MPI presentation

Example of problems 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();
```

Example of problems with MPI

Token circulation while computing on 4 nodes

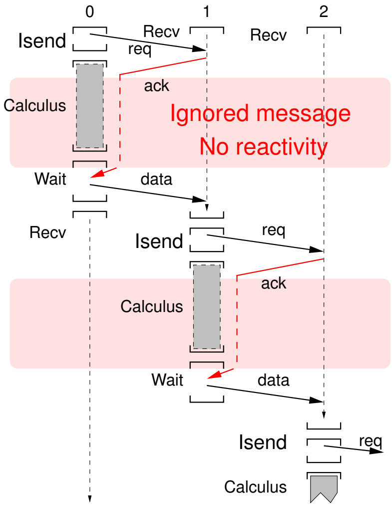
```
if (mynode!=0)
  MPI_Recv();

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

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

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

Example of problems 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

Improving MPI reactivity

Possible solutions

- add calls to `MPI_test()` in the code
- using a multithreaded MPI version
  - \+ parallelism, communication progression independent from computations
  - \- busy waiting synchronisation less efficient
  - \- scrutation must be managed

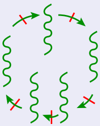
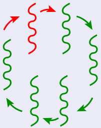
Integrate a scrutation server into the scheduler

Scheduler: required for optimal behaviour

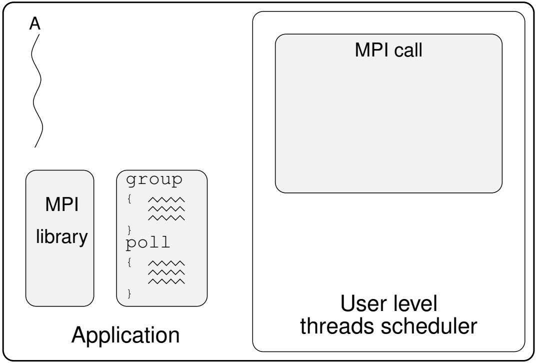
- system is known by the scheduler
  - it can choose the best strategy to use

Efficient and reactive scrutation

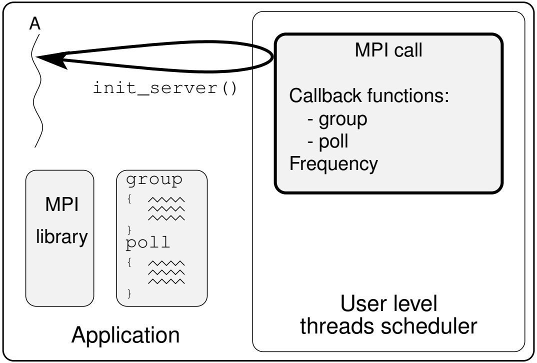
- less context switches
- guarantee frequency
  - independent with respect to the number of threads in the application


instead of


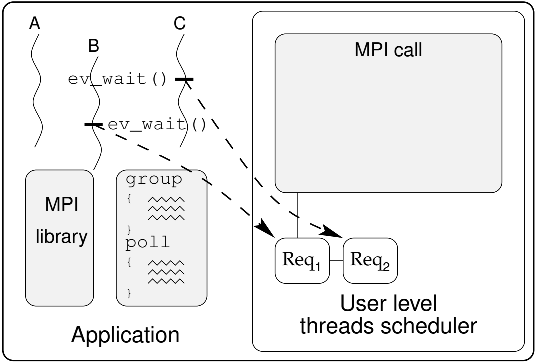
Running the scrutation server



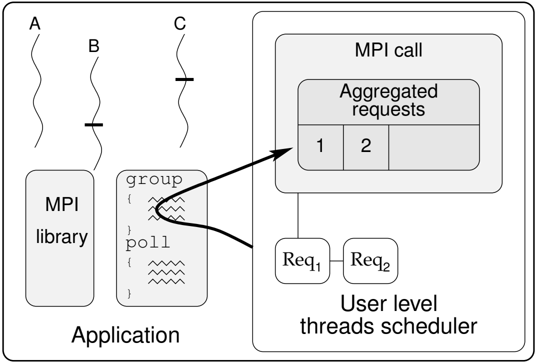
Running the scrutation server



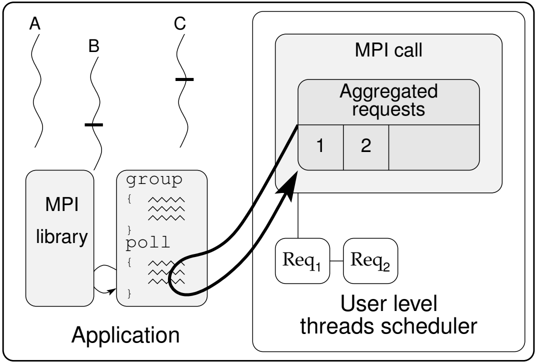
Running the scrutation server



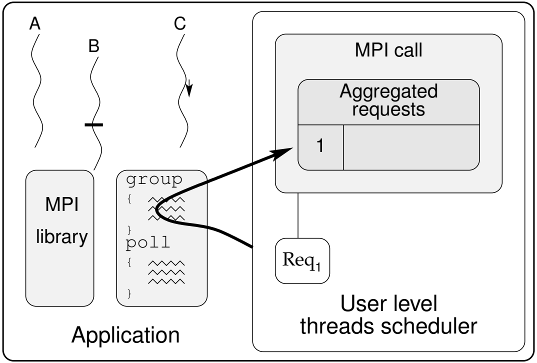
Running the scrutation server



Running the scrutation server



Running the scrutation server



Part V

Conclusion

Outlines: Conclusion

Conclusion

Multi-threading

- cannot be avoided in current HPC
- directly or through languages/middlewares
- difficulties to get a efficient scheduling
  - no perfect universal scheduler
  - threads must be scheduled with respect to memory (NUMA)
  - threads and communications must be scheduled together