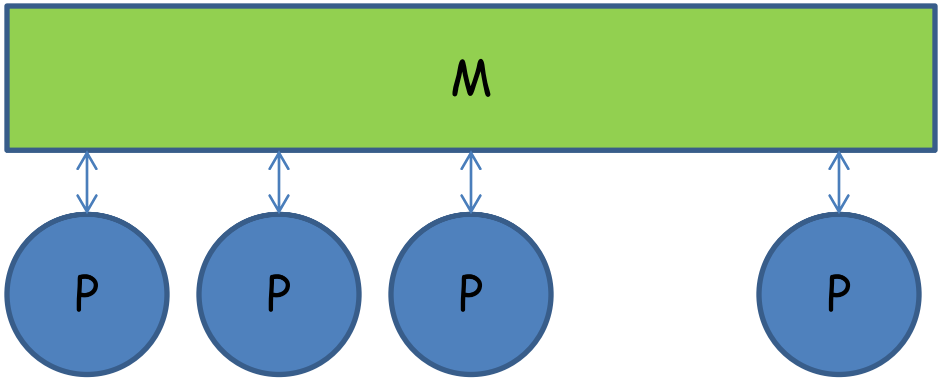
Introduction to Parallel Computing

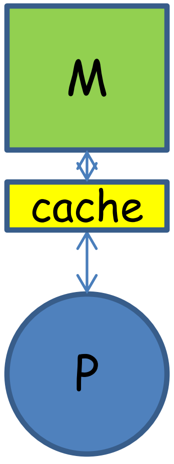
Jesper Larsson Träff
Technical University of Vienna
Parallel Computing

Shared-memory architectures & machines



Shared-memory model

Naive, shared memory (programming) model: processors execute processes, processes are not synchronized, special methods for sharing memory between processes, NUMA



Cache: small, fast memory, close to processor, accessed main memory locations are stored temporarily in cache, reused when possible

Caches may help to alleviate/hide memory („von Neumann") bottlenect

- Main memory: Gbytes, access times > 100 cycles
- Cache: Kbytes->Mbytes, access times,1-20 cycles

Typically 2-3 levels of caches in modern processors, and several special caches, TLB, victim cache, instruction cache, …

Caches, recap.

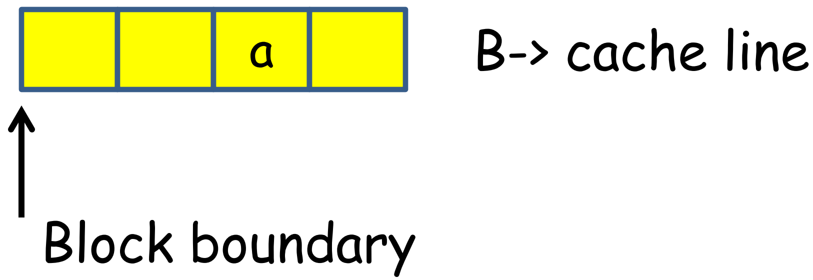
Cache consists of a number of lines that stores blocks of memory. A cache line holds a block and additional status information (dirty/valid bit, tag)

Typical block size: 64Bytes

Caches exploit and makes sense because of:
- Temporal locality: locations are typically used several times in close succession, several operations on same operand
- Spatial locality: when a location is addressed, typically locations close to it (a+1, a+2, …) will be also be used

Properties of algorithms/programs, and not always so

Access to main memory in block size units B, aligned to block boundary



Memory read a:
if address a already in cache, reuse from there, if not read from memory through cache, evict previous line

Memory write a:
different possibilities. If a is already in cache, write overwrites; if a is not in cache

- Write allocate: if a is not in cache, read a
- Write non-allocate: write directly to memory

- Write-through cache: each write is immediately passed on to memory (typically non-allocate)
- Write back: cache line block is written back when line is evicted (typically write allocate)

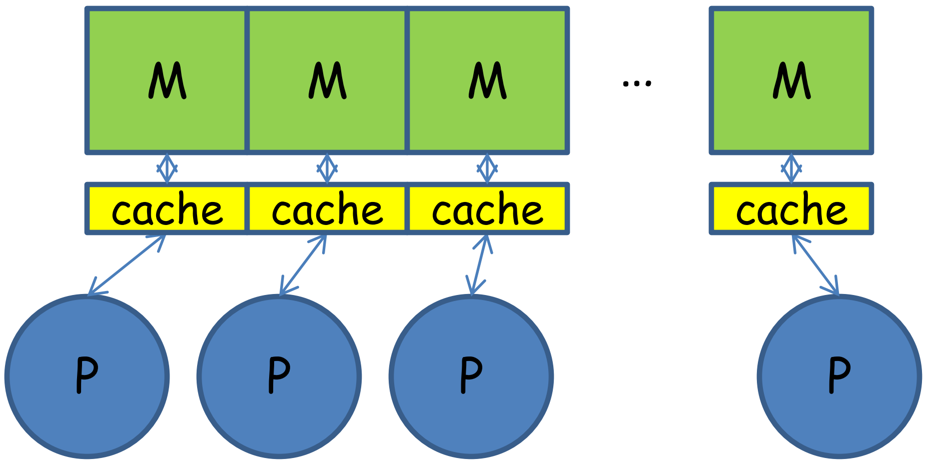
Address a:
•If a can go into only one specific line of the cache: directly mapped
•If a can go into any line of the cache: fully associative

•If a can go into any of a small set of lines: set-associative (typically 2-way, 4-way)

Replacement policies for associative caches
•LRU: least recently used
•LFU : least frequently used

Typically, all maintained in hardware

Multiprocessor/multi-core caches



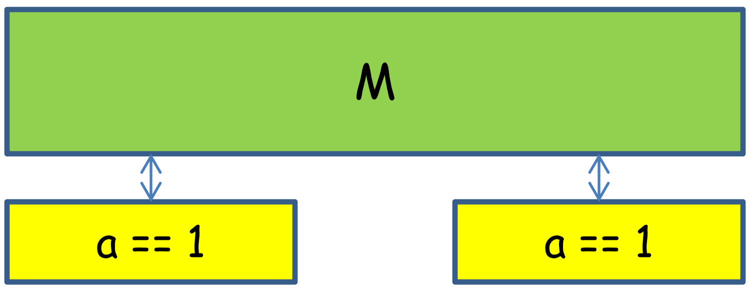
Shared-memory model, cc-NUMA

Cache-coherent non.uniform memory access

Typically, several cores shares caches at some levels

Cache coherence

Processor/core 0 and 1 with private caches, both have read location a into cache. Processor 0 writes to a?



a =7;

b = a; // ??

Read by 1 occurs „after" write by 0. If b is still 1, cache system is not coherent

Let the order of memory accesses to a specific location a be given by the program order

Cache is coherent if

1. If processor P writes to a at time t1 and reads a at t2>t1, and there are no other writes (by P or other) to a between t1 and t2, then P reads the value written at t1
2. If P1 writes to a at t1 and another P2 reads a at t2>t1 and no other P writes to a between t1 and t2, then P2 reads the value written by P1 at t1
3. If P1 and P2 writes to a at the same time, then either the value of P1 or the value of P2 is stored at a

Ad 1. Program order is preserved for each processor for locations that are not written by other processors

Let the order of memory accesses to a specific location a be given by the program order

Cache is coherent if

1. If processor P writes to a at time t1 and reads a at t2>t1, and there are no other writes (by P or other) to a between t1 and t2, then P reads the value written at t1
2. If P1 writes to a at t1 and another P2 reads a at t2>t1 and no other P writes to a between t1 and t2, then P2 reads the value written by P1 at t1
3. If P1 and P2 writes to a at the same time, then either the value of P1 or the value of P2 is stored at a

Ad 2. Here, t1 and t2 have to be „sufficiently" separated in time. Updates by P1 must eventually become visible to the other processors

Let the order of memory accesses to a specific location a be given by the program order

Cache is coherent if

1. If processor P writes to a at time t1 and reads a at t2>t1, and there are no other writes (by P or other) to a between t1 and t2, then P reads the value written at t1
2. If P1 writes to a at t1 and another P2 reads a at t2>t1 and no other P writes to a between t1 and t2, then P2 reads the value written by P1 at t1
3. If P1 and P2 writes to a at the same time, then either the value of P1 or the value of P2 is stored at a

Ad 3. Writes are required to „serialize". Either of the values simultaneously written will be stored. „Same time" means „sufficiently close" in time.

cc-NUMA systems (most multi-core and SMP nodes): cache coherent, non-uniform memory access

Cache coherence maintained by hardware at the cache line level. Standard approaches and protocols:

- Update based
- Invalidation based

- Snooping/bus based
- Directory based

All: expensive in hardware („transistors“, „power“); can affect performance negatively

Sharing/false sharing

Cache coherence is maintained at the cache line level. Processor 0 updates y, processor 1 updates x (with e.g. &x == &z[1], &y = &z[2])


```
for (i=0; i<n; i++) y += i-1;
```

```
for (i=0; i<n; i++) x += 2*i;
```

Although x and y are different memory locations, each update will cause cache coherency traffic!! Because cache coherency is at the cache line level, x and y are falsely shared

Memory consistency

In what order do writes to different locations not necessarily in cache become visible in memory and to other processors?

Core 0:

```
x = 0;
// … some code
x = 1;
if (y==0) {
  // body
}
```

Core 1:

```
y = 0;
// … some code
y = 1;
if (x==0) {
  // body
}
```

x not in cache of core 1, y not in cache of core 0

Can core 0 and core 1 both execute body of if-statement?

Core 0:

```
x = 0;
// … some code
x = 1;
if (y==0) {
  // body
}
```

Core 1:

```
y = 0;
// … some code
y = 1;
if (x==0) {
  // body
}
```

If x=1; y=1; appears at the same time, no cores execute body

If core 0 in body, then core 1 has executed y=0; but not y=1; thus core 1 cannot enter body

Correct?

Only under sequential consistency (or similar)

Sequential consistency: memory accesses of each processor are performed in program order; program result is as for some interleaving of the memory accesses of all processors

Sequential consistency is typically not guaranteed by modern multiprocessors:

- Caches, may delay writes
- Write buffers, may delay and/or reorder writes
- Memory network: may reorder writes
- Compiler: may reorder updates

Relaxed consistency models (see other lecture…) pose weaker constraints on hardware, may still allow correctness reasoning

In short:
To guarantee intended effect/correctness of a shared-memory multiprocessor program, special instructions that enforce memory updates to take effect may have to be used

Example:

memory fence(f) : completes all writes before the instruction and sets flag f

Another processor waiting for f will „know" that all writes of the other processor before f was set will have been completed

Other approaches to alleviating memory bottleneck

•Prefetching: start loading operands well before use

•Multi-threading: when a thread („virtual processor“) issues a load, switch to another thread

Note: multi-threading requires explicitly parallel programs

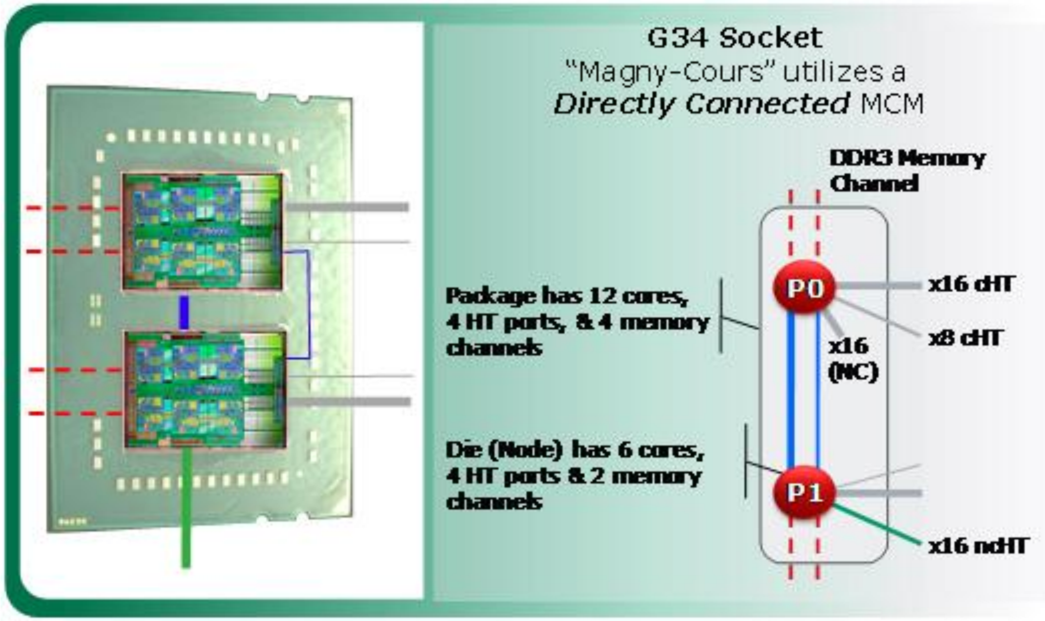
Note: both prefetching and multi-threading are latency hiding techniques. Memory bandwidth is still required for the number of outstanding memory requests

TU Wien parallel computing shared-memory node

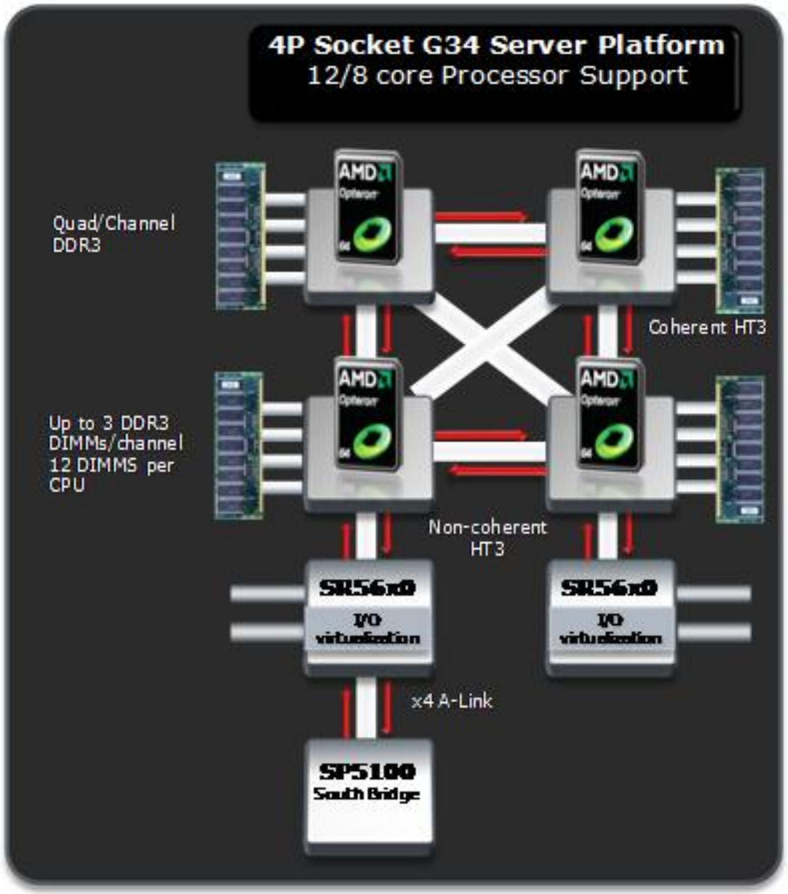
4xAMD „magny cours“ 12-core Opteron 6168 processors
128GByte main memory, 1.9GHz

- Per core L1 cache: 128KB
- Per core L2 cache 512KB
- Shared L3 cache 12288KB

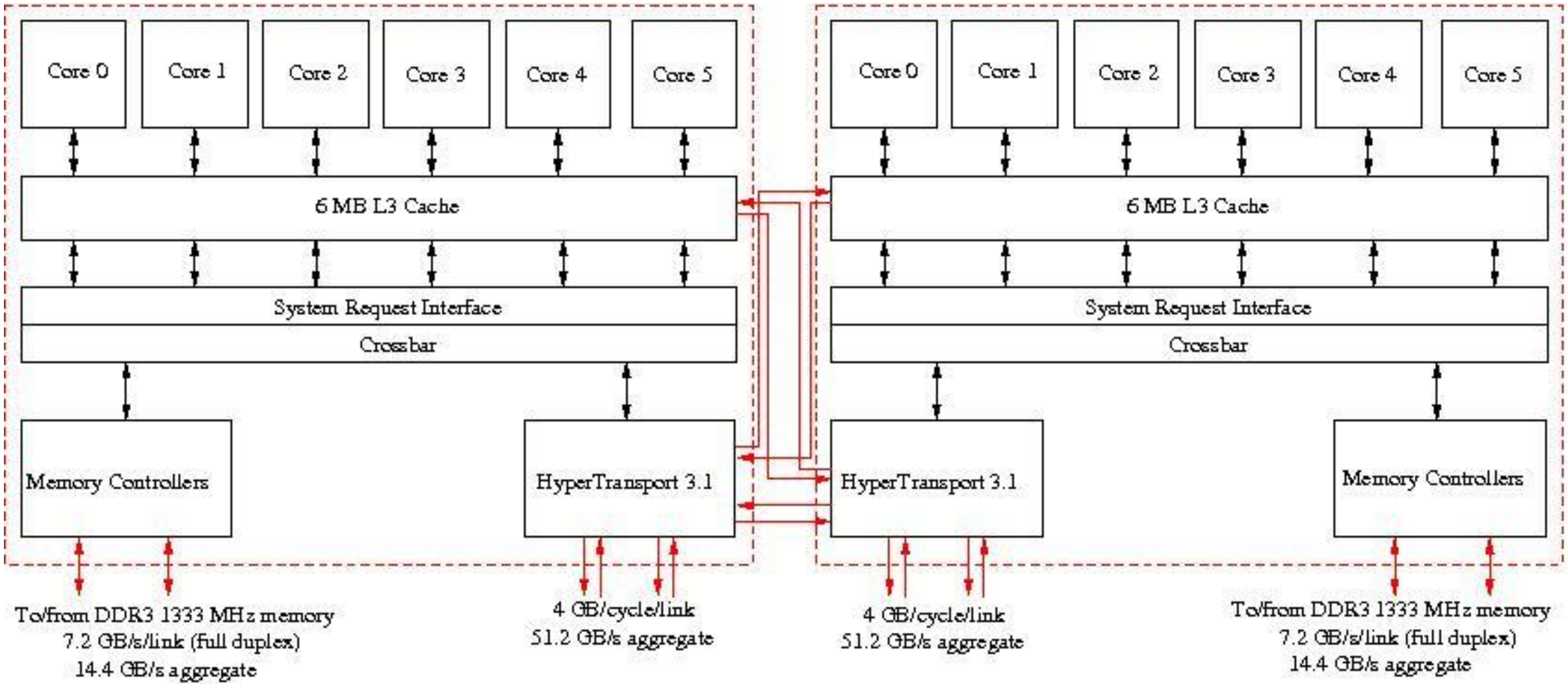
12 core = 2x6 cores, 2 dies on chip?



HT: HyperTransport – standardized processor-processor interconnect

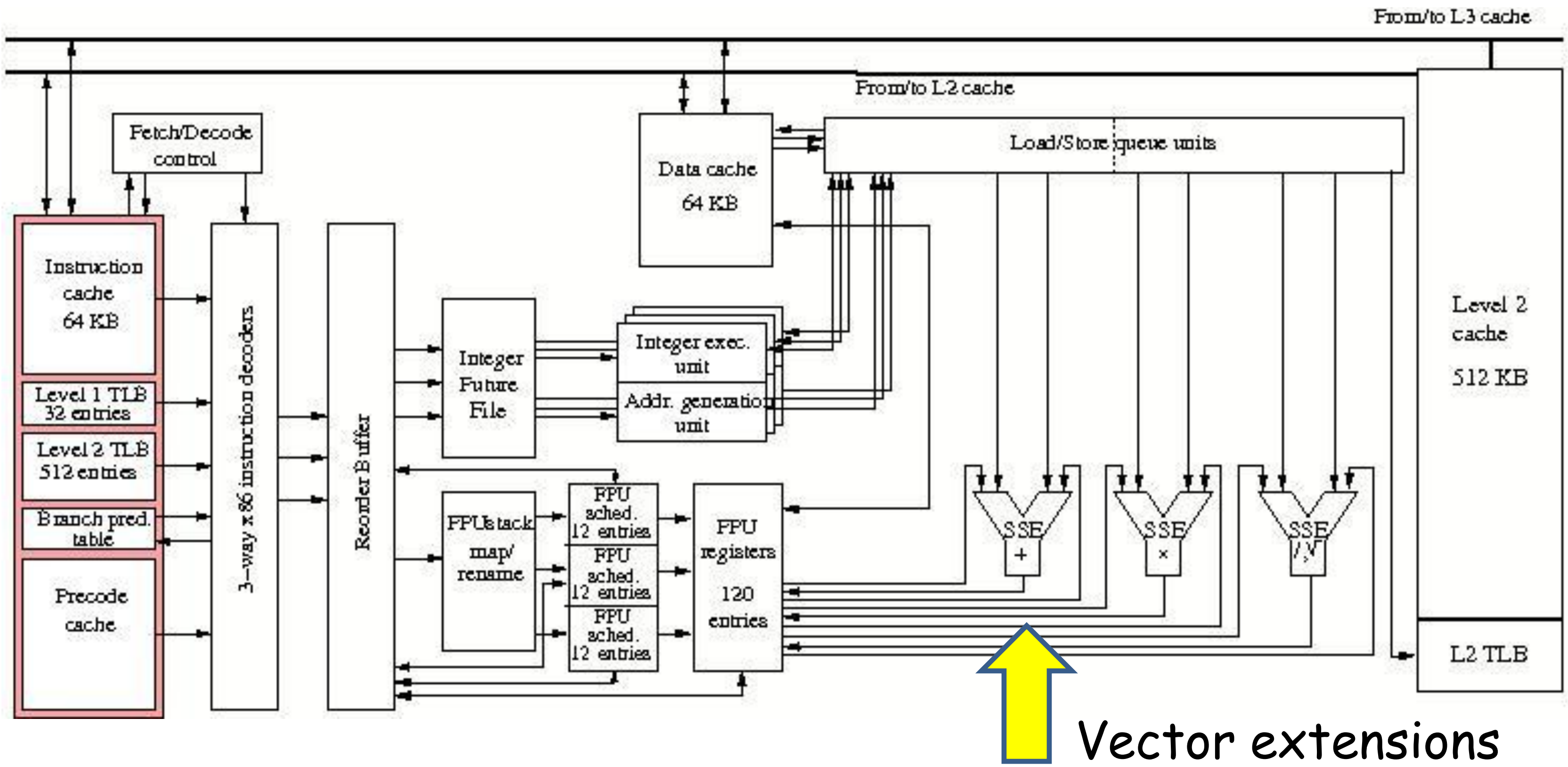


48-core shared-memory system from4x12-core



Check-exercise: try to find the (superscalar) issue width? Peak performance? of the Opteron/Magny Cours processor

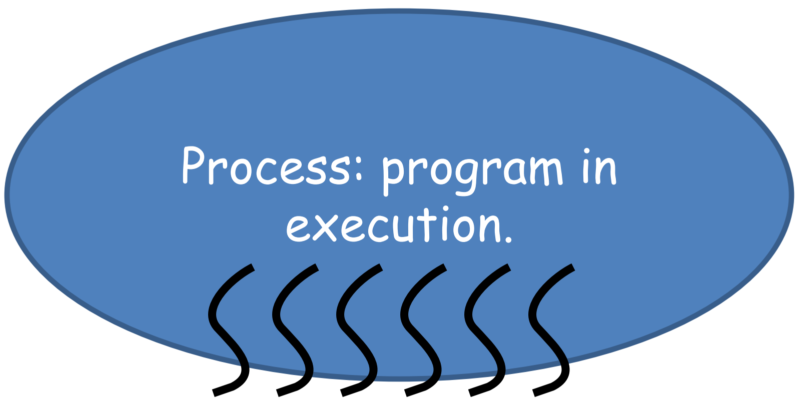
From University of Utrecht, EuroBen homepage: www.phys.uu.nl/eurben



L1 cache: 64KB data, 64KB instruction

Thread model

Thread: independent stream of instructions that can be scheduled by the OS. Typically, threads live inside an OS „process", and shares all global information of the process (Thread: „smallest unit that can be independently scheduled")



UNIX process global information:
- File pointers
- Global variables
- Static variables
- Heap storage

Per thread: local variables (stack), registers, „thread local storage"

POSIX threads, pthreads

POSIX: Portable Operating Systems Interface for uniX

Standard thread library API for UNIX (Linux etc.) since 1995: IEEE/ANSI 1003.1c-1995

Official standard documents cost money; standard available as man pages, internet, several tutorials and books

Low-level interface for C/UNIX thread programming

More modern thread model, including memory model: Java threads

(p)threads „Programming model"

1. Fork-join type parallelism: a thread can „spawn" (start) any number of new threads (up to system limitations), wait for threads to terminate

2. Initially one main („master") thread is active. Code for thread is a procedure/function

3. Spawned threads are peers, any thread can wait for termination of any other thread

4. Threads are scheduled by the underlying system, may or may not run simultaneously, may or may not be scheduled to available processors/cores

5. No implicit synchronization between threads, threads progress independently, and asynchronously

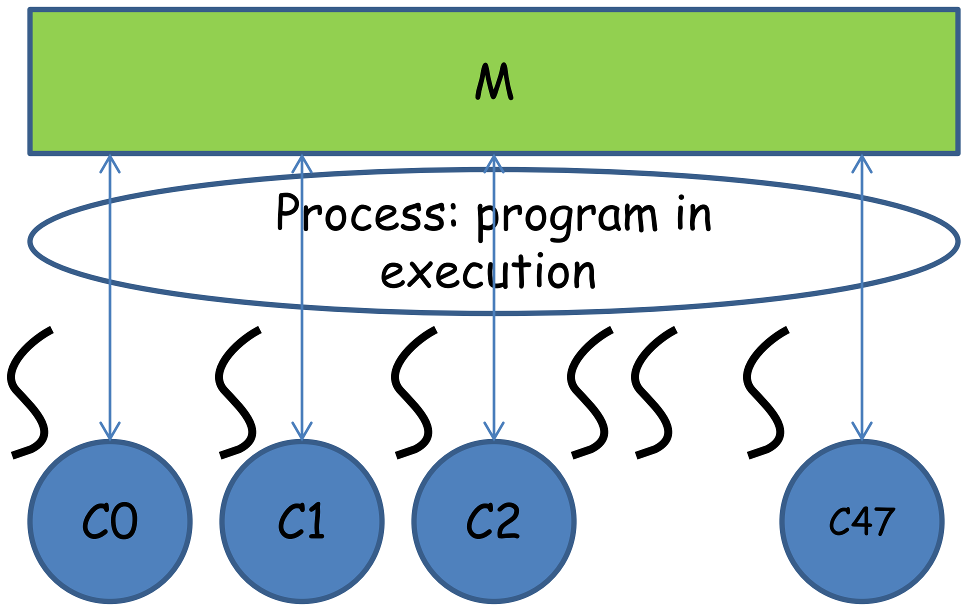
6. Threads share process global data

7. Coordination mechanisms for protecting access to shared variables (locks, condition variables). All concurrent updates must be protected, otherwise program illegal, outcome undefined

8. …

Pthreads: no cost model, no memory model, …

Pragmatics (for parallel computing): runable threads are expected to be scheduled to free cores. Scheduling and binding (mapping to specific core) can be influenced



pthreads for C:

Main program is main thread, spawns off and waits for termination of additional threads. Thread code: C function

- Include header `<pthread.h>`
- All pthread types and functions prefixed by `pthread_`
- pthread functions return error code, or status information, good practice to check!! (not done here…)

Compile with

```
gcc -Wall -o pthreadshello pthreadshello.c -pthread
```

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

`pthread_t`: type of thread object (opaque), thread id returned here (pointer), must be allocated globally by spawning thread

```
static pthread_t newthread
```

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

`void *(start_routine)(void *)`: type template for the function to run as thread. Takes arguments via generic pointer, returns generic pointer, standard C convention

```
void *newcode(void *genericargs) {
  myarg_t realargs = (myarg_t*)genericargs;
  // work to be done by this thread
}
```

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

`void *`: pointer to arguments, must have beeen allocated by spawning thread in static memory (heap)

```
struct {
  // args
} *
```

Parallel Computing

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Execution of thread can be influenced by attributes: stacksize, scheduling properties, … NULL, or

Not this lecture

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Finalizing/terminating thread

```
#include <pthread.h>

void pthread_exit(void *status);
```

Terminates thread, pointer to return status can be supplied; return status can be caught by joining thread

Joining threads

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **status);
```

Main thread New thread Some other thread

```
int main () {
  pthread _t t;
    pthread_create(&t,...);
  ... // main continues


}
```

```
threadcode() {
   // ...
  pthread_exit(NULL);
}
```

```
pthread_join(t,NULL);
```

A small example

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

// pthreads header
#include <pthread.h>

// global state; here read-only - don't do this…
int threads_glob;

void *something(void *argument){
  int rank = (int)argument;

  printf("Thread rank %d of %d responding\n",
         rank,threads_glob);
  pthread_exit(NULL);
}
```

C style: cast void * argument back to intended type

Parallel Computing

A small example

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

// pthreads header
#include <pthread.h>

// global state; here read-only - don't do this…
int threads_glob;

void *something(void *argument){
  int rank = (int)argument;

  printf("Thread rank %d of %d responding\n",
         rank,threads_glob);
  pthread_exit(NULL);
}
```

Here misuse of pointer to store rank

Parallel Computing

```
int main(int argc, char *argv[]){
  int threads, rank;
  int i;  pthread_t *handle;

  threads = 1;
  for (i=1; i<argc&&argv[i][0]=='-'; i++) {
    if (argv[i][1]=='t')
      i++,sscanf(argv[i],"%d",&threads);
  }
  threads_glob = threads;
  // number of threads read and stored globally

  handle =
    (pthread_t*)malloc(threads*sizeof(pthread_t));
  // fork the threads
  for (i=0; i<threads; i++) {
    pthread_create(&handle[i],NULL,
                   something,(void*)i);
  }
```

Getting command line arguments

Local scalar variable cast into generic void pointer, correct, but dangerous misuse

TU WIEN

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

// pthreads header
#include <pthread.h>

// global state; here read-only - don't do this…
int threads_glob;

void *something(void *argument){
  int rank = *(int*)argument;

  printf("Thread rank %d of %d responding\n",
         rank,threads_glob);
  pthread_exit(NULL);
}
```

Better: cast and deref

Parallel Computing

```
int main(int argc, char *argv[]){
  int threads, rank;
  int i;  pthread_t *handle;

  threads = 1;
  for (i=1; i<argc&&argv[i][0]=='-'; i++) {
    if (argv[i][1]=='t')
      i++,sscanf(argv[i],"%d",&threads);
  }
  threads_glob = threads;
  // number of threads read and stored globally

  handle =
    (pthread_t*)m      (thre    *s zeof(pthread_t));
  // fork the th  ad
  for (i=0; i<th  ads;      )
   pthread_crea  &handl       N  L,
                            ,&i);
  }
```

Only one (local) variable, may be overwritten befor   as copied into local

Problem?

Example:
a value (storage of i) is overwritten by one thread, may (or may not) happen before the other threads have read intended value. Program outcome dependent on relative timing of threads. Bad, unintended non-determinism…

Race condition:
Outcome of parallel progam execution is dependent on the relative timing of the updates by processors/threads

```
int main(int argc, char *argv[]){
  int threads, *rank;
  int i;  pthread_t *handle;

  // … get the number of threads
  handle =
    (pthread_t*)malloc(threads*sizeof(pthread_t));
  rank = (int*)malloc(threads*sizeof(int));
  // fork the threads
  for (i=0; i<threads; i++) {
    rank[i] = i;
    pthread_create(&handle[i],NULL,
                   something,&rank[i]);
  }
  // join the threads again
  for (i=0; i<threads; i++) pthread_join(handle[i],NULL);
  free(rank); free(handle);
  return 0;
}
```

Own location for each thread, no overwrite

Wait for threads to terminate

Free storage nicely

Parallel Computing

#define NDEBUG
// assertion checking disabled

Checking return codes with assertions

Enables assertion checking, macro assert(expr);

```
#include <assert.h>

int main(int argc, char *argv[]){
  int threads, *rank;
  int i;  pthread_t *handle;

  // … get the number of threads, allocate

  // fork the threads
  for (i=0; i<threads; i++) {
    rank[i] = i;
    errcode = pthread_create(&handle[i],NULL,
                             something,&rank[i]);
    assert(errcode==0);
  }
  // …
}
```

Assertion `errcode==0` expected to evaluate to true (≠0), otherwise abort

Potential problem: sequential spawning of treads can limit scalability (Amdahl).

In general: thread creation can be expensive

```
for (i=0; i<threads; i++) {
    rank[i] = i;
    pthread_create(&handle[i],NULL,
                   something,&rank[i]);
  }
  // join the threads again
  for (i=0; i<threads; i++) pthread_join(handle[i],NULL);
```

Fix: spawn recursively

pthread_t thread identifiers are opaque; normally user gives thread „identity“ (as in example), a thread can inquire ist own pthread_t id; pthread_t id's can be compared

```
#include <pthread.h>

pthread_t pthread_self(void);
```

```
#include <pthread.h>

int pthread_equal(pthread_t thread_1,
                  pthread_t thread_2);
```

Explicit parallelization of data parallel loop

```
for (i=0; i<n; i++) {
   a[i] = f(i);
}
```

Thread i (on core i) performs

```
for (i=start; i<end; i++) {
   a[i] = f(i);
}
```

start = i*n/threads
end = (i+1)*n/threads

Explicit parallelization of data parallel loop

```
for (i=0; i<n; i++) {
   a[i] = f(i);
}
```

Arguments struct

```
typedef struct {
  int *array;
  // pointer shared, global data
  int start, end;
  int rank; // threads rank
} rankindex_t;
```

```
loopblock(void *what)
{
  rankindex_t *ix = (rankindex_t*)what;
  int *a = ix->array;
  int i, start=ix->start, end=ix->end ;

  for (i=start; i<end; i++) a[i] = f(i);
}
```

Function for loop block

Example: matrix-vector product

y= x*A, nxm matrix x, n vector A

```
for (i=0; i<n; i++) {
  y[i] = 0;
  for (j=0;j<m; j++) {
    y[i] += x[i][j]*A[j];
  }
}
```

Nested loop

Parallelized by tiling outer loop

```
for (i=rank; i<n; i+=threads) {
  y[i] = 0;
  …
```

Each thread rank handles every threads'th index

Thread rank:

```
for (i=rank; i<n; i+=threads) {
  y[i] = 0;
  for (j=0;j<m; j++) {
    y[i] += x[i][j]*A[j];
  }
}
```

Problem?

y[0]	= 0;
y[1]	= 0;
y[2]	= 0;
y[3]	= 0;

y values go into (local) caches

Parallel Computing

Thread rank:

```
for (i=rank; i<n; i+=threads) {
  y[i] = 0;
  for (j=0;j<m; j++) {
    y[i] += x[i][j]*A[j];
  }
}
```

y[0] += x[i][j]…;
y[1] += x[i][j]…;
y[2] += x[i][j]…;
y[3] += x[i][j]…;

False sharing: updates on y causes either cache update traffic or invalidates/memory reads

Thread rank:

```
for (i=rank*n/p; i<(rank+1)*n/p; i++) {
  y[i] = 0;
  for (j=0;j<m; j++) {
    y[i] += x[i][j]*a[j];
  }
}
```

Solution?

Exercise: test effects of false sharing (best and worst cases) on TU Wien parallel computing shared-memory node, with explicit thread affinity

Binding threads to cores

```
#define _GNU_SOURCE
#include <pthread.h>

int pthread_setaffinity_np(pthread_t thread,
                           size_t cpusetsize,
                           const cpu_set_t *cpuset);
Int pthread_getaffinity_np(pthread_t thread,
                           size_t cpusetsize,
                           cpu_set_t *cpuset);
```

`_np`: non-portable, non-standard extension to pthreads (but commonly supported in some form)

Thread will be migrated to one of the cores in cpuset

Coordination constructs for avoiding race conditions

- Locks/mutex'es – for ensuring mutual exclusion
- Condition variables
- Advanced, non-standard features: semaphores, barriers, spin locks

Note: these are all classical concurrent computing constructs. Some classical algorithms to solve the problems under weak architecture assumptions: Dekker's algorithm, Lamport's bakery, …

Caution: the constructs were developed for few resources, not necessarily sufficient for highly parallel, scalable programming

Critical section:
Code manipulating shared resources, that must not be concurrently manipulated by other active entities (threads, processes, …)

Shared resources: simple variables, data structures, devices

Mutual exclusion property/algorithm: at most one thread in given critical section

Pthread „model": it is not allowed to update shared variables outside of critical sections. The lock constructs shall ensure a consistent view of memory.

Locks

Lock: shared object between any number of threads.

Lock state: locked (acquired), or unlocked (released)

At most one thread can acquire the lock, must release after use. When a thread attempts to acquire a lock that is already acquired by another thread it is blocked, and waits until the lock is released

If any thread that is waiting to acquire a lock is eventually granted the lock, the lock is called fair!!

Pthread lock is called mutex, type `pthread_mutex_t`

Static allocation and initialization with

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Dynamically allocated mutexes

```
#include <pthread.h>

Int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutex_attr *attr);
```

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Locking and unlocking

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Unsafe program, what is the intended value of x for thread 0 and 1?

```
x = 0;
```

Thread 0:

```
a = x;
```

Thread 1:

```
b = x;
```

Thread 2:

```
x = c;
```

Race condition

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
lock(&lock);
a = x;
unlock(&lock);
```

Thread 1:

```
lock(&lock);
b = x;
unlock(&lock);
```

Thread 2:

```
lock(&lock);
x = c;
unlock(&lock);
```

Mutual exclusion enforced

Both read and write accesses to x need to be protected by the lock mutex

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
lock(&lock);
a = x;
unlock(&lock);
```

Thread 1:

```
lock(&lock);
b = x;
unlock(&lock);
```

Thread 2:

```
lock(&lock);
x = c;
unlock(&lock);
```

Mutual exclusion enforced

Note: pthread locks are not fair, no guarantee that a thread trying to acquire a lock will eventually acquire it

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
lock(&lock);
lock(&lock);
a = x;
unlock(&lock);
```

Thread 1:

```
lock(&lock);
b = x;
unlock(&lock);
```

Thread 2:

```
lock(&lock);
x = c;
unlock(&lock);
```

Deadlock!

Deadlock: two or more threads are in a situation where they dependently on each other cannot progress. Deadlock will eventually proliferate to all threads

What about this?

Thread 0:

```
a = f(x);
```

Thread 1:

```
b = f(x);
```

Thread 2:

```
c = f(y);
```

No apparent races, independent evaluation of some function f

OK?

Depends on f, must be such that it can indeed be executed concurrently: „tread safe“

Thread safety

Tautological definition: a function is thread-safe if it can be executed concurrently by any number of threads and will always produce correct results

Non-thread safe functions are

1. Functions that do not protect (write access) to shared variables
2. Functions that keep state over successive invocations (`static` variables).
3. Functions that return pointers to `static` variables
4. Functions that call thread-unsafe functions

Careful with functions supplied by other party, e.g. system functions

Example: `rand()` keeps state internally in static variables, notoriously not thread safe

Some system functions are made thread safe by locking. Can have undesirable effects – serialization slowdown, deadlock