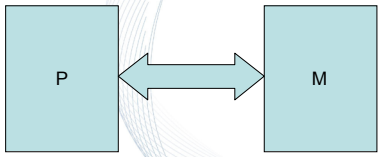
COURS PARALLELISME

Programming muticore architectures

Jean-Louis PAZAT INSA/IRISA

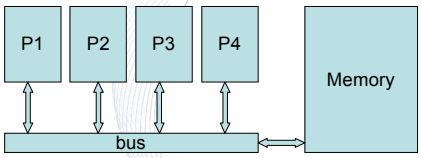
Computer architecture for Dummies

- Needed:
  - A processor (computation)
  - memory (storage)
  - I/Os (communication)

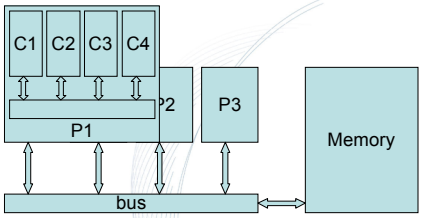


Multiprocessors

- Speeding up ?
  - Add processors



Multicores



Why multicore architectures ?

+ 1% freq	+ 3% heat	+0,66% perf
+15%	+45%	+9%
-30%	-90%	80%
-50%	-150%	66%
-15% (2 ♥)	2 * 45% ~ 1	2*90%=180%
-25% (4 ♥)	4* 25% ~1	4*82,5% = 330%

Why multicore archtectures ?

- Presentation from Charles Congdon (Intel)

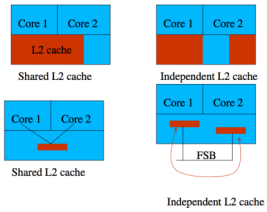
whymulticore.flv

What is a multicore architecture ?

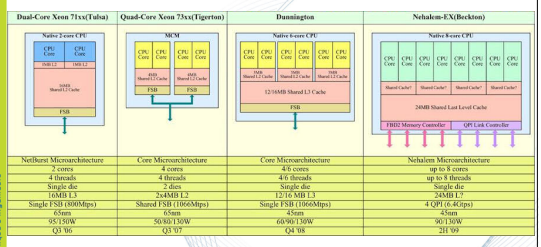
- Uniform multicore architectures
  - A « core » is the core of a processor :
    - ALU, registers, sequencer
    - 0,1, or 2 caches
    - 1 core can run one or more threads
  - A processor may have many cores
    - Past : 1 core
    - Now : from 2 to 8 cores
    - In a few years: > 100 cores
  - Cores share cache(s)
  - A multicore processor needs efficient memory access

Why sharing caches ?

- Best use of cache space
- Efficient ommunication between cores
- Example L2 cache sharing

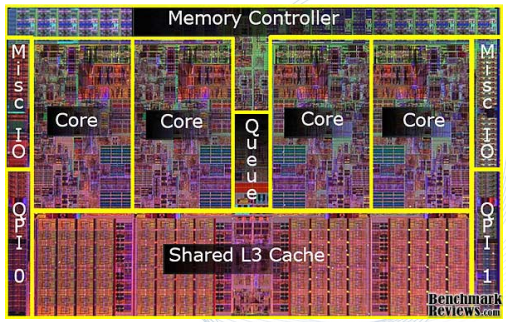


Les Intel multicore architectures
(outdated fig.)

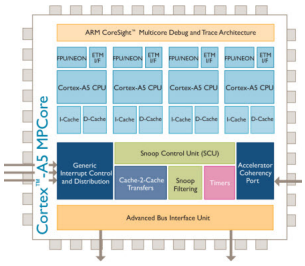


NetBurst Microarchitecture	Core Microarchitecture	Core Microarchitecture	Nehalem Microarchitecture
2 cores	4 cores	4/6 cores	up to 8 cores
4 threads	4 threads	4/6 threads	up to 8 threads
Single die	2 dies	Single die	Single die
16MB L3	2x4MB L2	12/16 MB L3	24MB L?
Single FSB (800Mtps)	Shared FSB (1066Mtps)	Single FSB (1066Mtps)	4 QPI (6.4Gtps)
65nm	65nm	45nm	45nm
95/150W	50/80/130W	68/90/130W	90/130W
Q3 '06	Q3 '07	Q4 '08	2H '09

Intel Nehalem



ARM multicore



Cell processors

- Heterogeneous chip multiprocessor
  - one 64-bit Power core
  - 8 specialized co-processors (SIMD) architecture
    - for data intensive processing
  - integrated by a coherent on-chip bus.
  - performance target
    - exploitation of parallelism through a high number of nodes on a chip multiprocessor.
  - Power reduce target
    - heterogeneous configuration with a SIMD-centered architecture.

Specific cores

- Cell processors

Top-level block diagram of the Cell Broadband Engine (CBE)

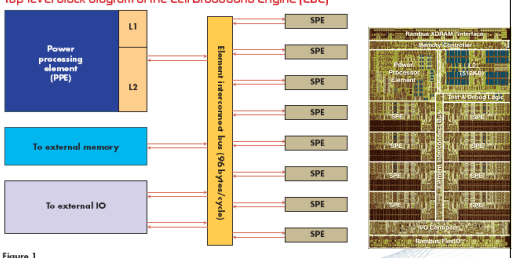
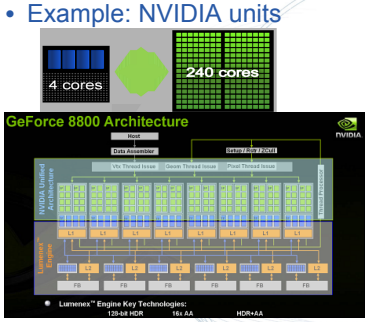


Figure 1

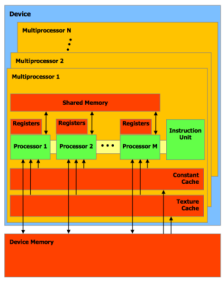
GP Units

- Example: NVIDIA units



GP/GPUs

- 1 or more GPUs on a board
- GPU Computing Tesla C870
  - 128 processors thread)
  - around 500 gigaflops
  - 1,5 Go of inboard memory



A CUDA example

- -> bureau

Institut National des Sciences Appliquées
École publique d'ingénieurs

Ancestors : SIMD machines ('70)

- Connection machine
  - 65536 processeurs, 4 Ko by processor
  - 1 sequencer:
    - At each cycle a processor
      - Executes the same operation as the others do
      - or
      - Does nothing
  - Fine for
    - Matrix/vector computation, image processing, …
    - Data parallelism

Institut National des Sciences Appliquées
École publique d'ingénieurs

SIMD programming

- Programming languages
  - Data parallel languages
    - Global opérations on data
      - alpha operations, beta reductions
      - « threads » for computation (CUDA)
    - Data movement
      - between processors (SHIFT, ROTATE, …) SUR CM
      - Between memory banks
    - Syntax
      - « Strange » or « C(++) -like »
  - Performance: do not forget :
    - SIMD = every proc does the same work or nothing
    - Programming language may be misleading

Institut National des Sciences Appliquées
École publique d'ingénieurs

Outline

- Multicore architectures
- Programming with Threads
- TBB
- Higher level languages
- openMP

Intel TBB

- TBB =
  - Threads Building Blocks
  - Intel C++ library
  - Uses natives threads
- Basic « object » : Task
  - Tasks are executed by native threads
  - Load balancing by Work stealing
- Templates to ease parallel programming
  - Parallel_for, parallel reduce, …
  - Synchronization tools (atomic, exmut,…)

Work stealing ?

- How to do load balancing ?
  - Centralized distribution of work
    - Master/worker, processor farms
  - Move work to under used resources
  - If you have not enough work steal work to another unit
    - Sound strange ?
    - Efficient (theoretically « optimal »)

```
void SerialApplyFoo(float a[],size_t n) {
  for( size_t i=0; i!=n; ++i )Foo(a[i])}
```

```
#include "tbb/tbb.h »
using namespace tbb;
class ApplyFoo {
 float *const my_a;
public:
 void operator()(const blocked_range<size_t>& r)
const {
  float *a = my_a;
for( size_t i=r.begin(); i!=r.end(); ++i )Foo(a[i]);}
ApplyFoo( float a[] ) :
my_a(a){}
};
```

```
void ParallelApplyFoo(float a[],size_t n)
  {parallel_for(blocked_range<size_t>(0,n),
  ApplyFoo(a));
```

OpenMP : présentation

OpenMP: une API pour écrire des Applications Multithread (spécification de 1997)

- Un ensemble de directives de compilation et une bibliothèque de fonctions
- Facilite la création de programmes multi-thread (MT) Fortran, C and C++
- Standardise la pratique de la programmation des machines SMP des 15 dernières années
- Plus facile que TBB mais pas C++

[F. Cappello, CNRS-LRI Orsay iHPerf2000 Aussois]

OpenMP : mode de programmation

OpenMP est utilisé « principalement » pour paralléliser les boucles :

Trouver les boucles les plus coûteuses en temps

Distribuer leurs itérations sur plusieurs threads.

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

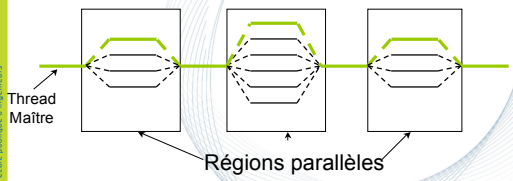
Programme Séquentiel

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Programme Parallèle

OpenMP : mode d ’exécution

- Parallélisme Fork-Join :
  - Le thread Maître lance une équipe de threads selon les besoins (région parallèle).
  - Le parallélisme est introduit de façon incrémentale ; le programme séquentiel évolue vers un prog. parallèle



L ’API OpenMP

- Les constructeurs OpenMP :
  - Les régions parallèles
  - La distribution du calcul
  - L’environnement de données
  - La synchronisation
- Bibliothèque de fonctions
- Variables d’environnement

OpenMP : Les régions parallèles

Les threads sont créés avec la directive “omp parallel”.

```
!$OMP PARALLEL [CLAUSE[[,] CLAUSE]...]
...
!$OMP END PARALLEL
```

Chaque thread exécute de manière redondante le code à l’intérieur du bloc structuré

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        fonc(ID,A);
}
```

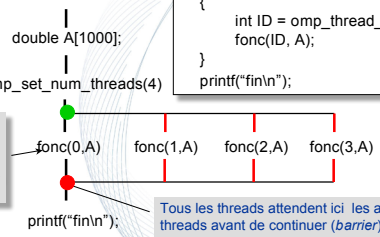
Tous les threads attendent les autres threads ici avant de continuer (*barrier*)

Chaque thread appelle fonc(ID,A) avec `ID` différent

OpenMP : Les régions parallèles

Tous les threads exécutent le même code.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        fonc(ID, A);
}
printf("fin\n");
```



Une seule copie de A est partagée entre tous les threads.

Tous les threads attendent ici les autres threads avant de continuer (*barrier*)

OpenMP : Synchronisations

- OpenMP possède les constructeurs suivants pour les opérations de synchronisation
  - `atomic`
  - `critical section`
  - `barrier`
  - `flush`
  - `ordered`
  - `single`
  - `master`

OpenMP : section critique

- Un seul thread à la fois peut entrer dans une section critique (`critical`)

```
!$OMP PARALLEL DO PRIVATE(B)
!$OMP& SHARED(RES)
      DO 100 I=1,NITERS
            B =  DOIT(I)
!$OMP CRITICAL
            CALL COMBINE (B, RES)
!$OMP END CRITICAL
100   CONTINUE
!$OMP END PARALLEL DO
```

OpenMP : action atomique

- **Atomic** est un cas spécial de section critique qui peut être utilisé pour certaines opérations simples.
- Elle s'applique seulement dans le cadre d'une mise à jour d'une case mémoire

```
!$OMP PARALLEL PRIVATE(B)
      B =  DOIT(I)
!$OMP ATOMIC
      X = X + B
!$OMP END PARALLEL
```

OpenMP : barrières de synchronisation

BARRIER tous les thread attendent que les autres threads arrivent au même point d 'exécution avant de continuer

```
#pragma omp parallel shared (A, B, C) private(id)
{
      id=omp_get_thread_num();
      A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
      for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
      for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
      A[id] = big_calc3(id);
}
```

Barrière implicite à la fin de la construction `omp for`

Barrière implicite à la fin de la région parallèle

Pas de barrière implicite `nowait`