Perspectives in parallel programming

MARCO DANELUTTO
PHD COURSE
MAY 2015
Perspectives in parallel programming

From sequential to parallel
- Many cores
- Xeon PHI
- Parallel
- ad-hoc many core
- FPGA
- data parallel
- custom algo (data flow)

From homogeneous to heterogeneous
- GPU
- Tiera

Introduction
- Performance
- Throughput
- Latency
- Datacenter (bill)
- Mobile (battery, yes/no)
- Fault tolerance
- Correctness
- Security
- Code

Non-functional features of interest
- Algorithmic skeletons
- Parallel design patterns
- Parallel building blocks
- Parallelism design phase
- Parallelism implementation phase
- Hardware targeting phase

Structured parallel programming methodology
- Rewriting
- Performance models
- Autonomic management of NF aspects
- RISC-pbb
- compiling to FPGAs
- DSL

Advanced features
Our world
From sequential to parallel

- Single core
  - Evolution of the concept of the processor
  - Single control flow

- Multi core
  - Evolution of the concept of CPU
  - Multiple control flows
  - Shared memory (coherent)

- Many core
  - Extreme evolution of the concept of CPU
  - Relaxing constrains (e.g. coherency, full interconnection, memory interfaces)
Single core

Von Neumann architecture
- Read instruction from MEM(PC), Interpret, Execute, Interrupts?

Pipeline architecture
- Interpose registers among stages

Superscalar architecture
- Replicate resources
Von Neumann Model

Von Neumann bottleneck
Pipeline architecture

Stage decoupling
- From clock to clock
  - Read registers, apply functions (delay), prepare data to be written in registers

- Number of stages
  - The higher, the shorter, the better (?)

- Pipeline flush
  - The longer, the worst
Superscalar Model

Resource replication
- Decode unit
- Execution unit
Code (pipeline/superscalar)

Single control flow

Dependencies
  ◦ Read after write on different units
  ◦ Misplaces in the pipeline

Branch penalties
Additional features

Branch prediction

Register reallocation

Out of order execution

...
Moore’s law

Components on chip double in 2 years

- From the ‘60
- Still valid somehow

Because

- Layout processing (down to O(10) nm)
- Better design (TTL, MOS, CMOS, ...)
- ...

-
In addition

Frequency
- Depends on the density and kind of components
- From MHz (1Mhz Z80, 8bit, 1980)
- To GHz (6GHz demonstrated in early ‘2000)

Memory
- More and more dense
- Not proportionally faster
- From KBytes to GBytes and further
Parallelism?

Pipeline
- $Ts = \max\{Tsi\}$

Superscalar (farm)
- $Ts = \frac{Tw}{Nw}$

$Tc \approx N_{\text{task}} \times Ts$
Memory parallelism

Interleaved memory
- Read blocks vs. words

Cache subsystem
- IC + DC
From sequential to parallel

- **Single core**
  - Evolution of the concept of the processor
  - Single control flow

- **Multi core**
  - Evolution of the concept of CPU
  - Multiple control flows
  - Shared memory (coherent)

- **Many core**
  - Extreme evolution of the concept of CPU
  - Relaxing constrains (e.g. coherency, full interconnection, memory interfaces)
Power wall

Power consumed proportional to the area

Computing power proportional to the square root of the area

\[ \sqrt{44} \]
Trend change

Formerly: more components for the single core
Now: more cores per CPU
Challenges (engineers!)

Core-core interconn
◦ Full, regular, ...

Memory model
◦ Cache coherent, protocols

Memory interface(s)
◦ Access off chip M
Challenges (computer scientists)

Radical change
- Need more and more control flows
- Possibly independent (or quasi independent)

Multi context core
- Introduce more needs
- Power 8 : 12 cores, 8 contexts per core, 4 sockets per board =384 threads !!!
From sequential to parallel

- **Single core**
  - Evolution of the concept of the processor
  - Single control flow

- **Multi core**
  - Evolution of the concept of CPU
  - Multiple control flows
  - Shared memory (coherent)

- **Many core**
  - Extreme evolution of the concept of CPU
  - Relaxing constrains (e.g. coherency, full interconnection, memory interfaces)
Intel Xeon PHI

60(1) cores per chip
x86 compatible
512bit vector unit
4 context per core
Tilera (TilePro64)

64 cores
Mesh interconnect
Programmable cache coherency protocol
(network processing)
Parallela Epiphany

<table>
<thead>
<tr>
<th>Technical info for</th>
<th>E16G301</th>
<th>E64G401</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>Core MHz</td>
<td>1000</td>
<td>800</td>
</tr>
<tr>
<td>Core GFLOPS/s</td>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>&quot;Sum GHz&quot;</td>
<td>16</td>
<td>51.2</td>
</tr>
<tr>
<td>Sum GFLOPS</td>
<td>32</td>
<td>102</td>
</tr>
<tr>
<td>mm$^2$</td>
<td>8.96</td>
<td>8.2</td>
</tr>
<tr>
<td>nm</td>
<td>65</td>
<td>28</td>
</tr>
<tr>
<td>W def.</td>
<td>0.9</td>
<td>1.4</td>
</tr>
<tr>
<td>W max.</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
From homogeneous to heterogeneous

GPUs
- Computer graphics

GP-GPUs
- Computer graphics and data parallel processing

FPGAs
- Anything in hardware
GP-GPUs

Co-processors
◦ Distinct memory (now merge, with otherheads)

Multiprocessors
◦ With cores (ALUs with memory access)

Only good for data parallel computations
Huge effort to program
GP-GPU programming

Explicit memory management
- Global, texture, registers

Explicit data and control transfer
- Memory copies
- Kernel execution

Proprietary/dedicated control
- Warps, threads

CUDA, OpenCL, ...
Transpose (CUDA)

The naïve transpose:

```c
__global__ void transposeNaive(float *odata, float* idata,
                                 int width, int height, int nreps)
{
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;
    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i] = idata[index_in+i*width];
        }
    }
}
```

<table>
<thead>
<tr>
<th>Effective Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048x2048, GTX 280</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Loop over kernel</th>
<th>Loop in kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Copy</td>
<td>96.9</td>
<td>81.6</td>
</tr>
<tr>
<td>Naïve Transpose</td>
<td>2.2</td>
<td>2.2</td>
</tr>
</tbody>
</table>
__global__ void transposeFineGrained(float *odata,
    float *idata, int width, int height, int nreps)
{
    __shared__ float block[TILE_DIM][TILE_DIM+1];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index = xIndex + (yIndex)*width;

    for (int r=0; r<nreps; r++) {
        for (int i=0; i < TILE_DIM; i += BLOCK_ROWS) {
            block[threadIdx.y+i][threadIdx.x] =
                idata[index+i*width];
        }
    }
    __syncthreads();

    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS) {
        odata[index+i*height] =
            block[threadIdx.x][threadIdx.y+i];
    }
}

<table>
<thead>
<tr>
<th>Effective Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048x2048, GTX 280</td>
</tr>
<tr>
<td>Loop over kernel</td>
</tr>
<tr>
<td>Loop in kernel</td>
</tr>
</tbody>
</table>

Simple Copy

| Naive Transpose          | 2.2  | 2.2  |
| Coalesced Transpose      | 16.5 | 17.1 |
| Bank Conflict Free Transpose | 16.6 | 17.2 |
| Fine-grained Transpose   | 80.4 | 81.5 |
| Coarse-grained Transpose | 16.7 | 17.1 |

__global__ void transposeCoarseGrained(float *odata,
    float *idata, int width, int height, int nreps)
{
    __shared__ float block[TILE_DIM][TILE_DIM+1];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    for (int r=0; r<nreps; r++) {
        for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
            block[threadIdx.y+i][threadIdx.x] =
                idata[index_in+i*width];
        }
    }
    __syncthreads();

    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
        odata[index_out+i*height] =
            block[threadIdx.y+i][threadIdx.x];
    }
}
However...

GP-GPUs provide substantial computing power
With reasonable power consumption
At a price of program complexity
FPGA

Field programmable gate array

Can implement
- Sequential networks
- Functions
- Data flow graphs
- ...

Logics Block
Interconnection
Input/Output
Switch Box
Connect Block
Using FPGAs
Programming FPGAs

By hand (impossible)

RTL

◦ Verilog, VHDL

Tools

◦ Vivado, ISE, Quartus, ...
module sample_moduleA(
    output C,
    input A, B
);

    wire w1;
    xor (w1, A, B);
    and (C, A, w1);
endmodule

module sample_moduleB(
    output D,
    input E, F
);

    wire G, H
    sample_moduleA SA1(G, E, F), SA2(D, G, E);
endmodule
Typical FPGA usage (coprocessor)

Program «in hardware»
- Interface (PCIe) (IP!)
- Business logic (RTL!)

Interface code with hardware
- Similar to GP-GPU
- With different language(s) & tools
More advanced devices

Intel ATOM 600

- Atom core
- Altera FPGA
  - Through PCIe
- Targeting advanced apps
  - Graphics, vision, military, ...
More advanced devices

XILINX Zynq
- Dual core ARM
- XILINX 7 FPGA
- No PCIe bus
- AXI/AMBA interconnect
Non functional features of interest

Performance
Power
Correctness
Fault tolerance
Security

Functional (business logic)

Non functional
Performance

Latency
  ◦ Shorten time to complete 1 task

Service time
(Throughput)
  ◦ Shorten time to compute M tasks
  ◦ Not necessarily the single one

Radically different goals and consequent strategies
Power

When running on batteries
  ◦ On/off

In a datacenter
  ◦ $$$

But also
  ◦ Power dissipation
  ◦ Dark silicon

_Dark silicon_ is a term used in the electronics industry. In the nano-era, transistor scaling and voltage scaling are no longer in line with each other, resulting in the failure of _Dennard scaling_. This discontinuation of Dennard scaling has led to sharp increases in power densities that hamper powering-on all the transistors simultaneously at the nominal voltage, while keeping the chip temperature in the safe operating range. "Dark Silicon" refers to the amount of silicon that cannot be powered-on at the nominal operating voltage for a given thermal design power (TDP) constraint. According to recent studies, researchers from different groups have projected that, at 8 nm technology nodes, the amount of Dark Silicon may reach up to 50%-80% depending upon the processor architecture, cooling technology, and application workloads. Dark Silicon may be unavoidable even in server workloads with abundance of inherent client request-level parallelism.
Correctness

Hardware & **software**
- Model checking parallel software
- ... 

Proving properties *by composition*

Testing

Integration

...
Fault tolerance

Need to deal with a number of different failures
- Survive to faults
- Recovery strategies
  - Reactive
  - Proactive

Some masked at the hardware level
- Cell with 6 cores (out of 8)
- Power PC core switchoff
HiPEAC vision 2015 challenges

Dependability by Design

Cyber-Physical Systems, the Internet-of-Things are all developments requiring the highest possible levels of security, safety and reliability for their adoption. In particular, security has to become one of the primary design features of the whole system; systems have to be dependable by design.
HiPEAC vision 2015 challenges

Managing System Complexity

In information science, we are entering the era of system of systems, with the accompanying exponential growth in complexity. We need new system paradigms, such as reactive systems and feedback systems to manage this huge increase in total complexity.
HiPEAC vision 2015 challenges

Energy efficiency

We need to overcome energy as the limiting factor for performance increase of systems. Instead of over-designing systems, targeting for best effort, we need design methodologies and tools that allow for systems to scale to the desired quality of service. Instead of adding system layers we need to investigate methods that allow for cross layer information flow and optimization. Instead of being locked-in in silicon-based Von-Neumann architectures, we need to investigate other architectures, such as neural networks, Bayesian systems, in non-silicon technology.
HiPEAC vision 2015 challenges

Entanglement between the physical and virtual world

Information systems will sense, monitor, and even control the physical world with Cyber-Physical Systems and the Internet-of-things. Dealing with large amounts of unstructured data will require new approaches to bridge the gap between the physical world and computer systems. As these systems also exercise control on the physical world, safety is a primary concern and becomes a primary design constraint.
HiPEAC envisioned solutions

**Challenge:** manage the complexity of systems that are increasing in size, distribution, parallelism, compute power, communication, connectivity, and amount of processed data.

**Solutions:**

- Develop new advanced Models of Computation, Abstractions, and Computing Paradigms to solve the software crisis.
- Explore the path of **reactive systems**, in particular by applying knowledge from the cybernetics domain concerning the use of **feedback loops** to stabilize dynamic complex system.
- Develop tools to help developers cope with the complexity of systems, especially with non-functional requirements (timing, power, reliability, ...).
- Further develop design space exploration tools that help selecting an optimum in a multi-dimensional solution space.
- Develop tools and approaches to validate complex systems composed of black or grey box components.
- Develop tools to design and test the next generation of hardware platforms, accelerators, 2.5D and 3D integrated systems, trusted modules, ...
- Enable expressing the concurrency in an application (the “what”) to improve mapping applications to parallel or distributed systems: compilers and tools are becoming better than humans (at least in specific domains) at this kind of transformations (the “how”). This could lead to more declarative programming (as opposed to imperative programming).
Declarative Programming

What

How

Imperative

Declarative
Parallelism (fundamentals)

Parallelism
  • What’s parallelism

Measures of interests
  • How can I evaluate parallel implementations

Parallelism kinds
  • Different perspectives in parallelism

Activity graph
  • Parallelism howto
What’s parallelism

Identify sub computations
  ◦ With no dependencies

To be executed on separate execution engines
  ◦ Concurrently / in parallel / in a distributed way / ...
  ◦ Sharing time slices

Computing the same result of the sequential computation
Measures of interest

Simple measures
- How long is taken to compute
- How often I can produce a result

Derived measures
- How much I’m faster
- How efficiently I use resources
Simple measures

Latency
- Time spent to compute something
  - From input availability to output availability
  - Single task performance

Service time
- Interval between two consecutive task delivery
  - Multiple task performance

Completion time
- Latency in the large
  - Time to compute several tasks, from first task input to last result output
Simple measures

Approximated values

- $T_c = m \times T_s$
- Completion time is the product of service time by task no
- E.g. pipe (see above)
Derived measures

Speedup

- Best sequential over parallel (n)
  - Best (!) sequential
  - Measures how better I am w.r.t. sequential

- Maximum speedup (n) is n

- Amdahl law
  - Max speedup is 1/f (f is the serial fraction)

\[ sp(m) = \frac{T_{seq}}{T(m)} \]

4 proc \( sp(4) = 5 \)
Derived measures

Efficiency
- Ideal parallel time(n) over parallel time(n)
- Speedup(n) over n

Measures how good I’ve been to exploit available resources
- Max efficiency is 1 (100%)
- Seq efficiency is always 1
Derived measures

Scalability
- Parallel(1) over parallel(n)
- Selfie ...
- Perfect scalability does not imply good speedup
Kind of parallelism

Data parallelism
- Items from a collection
- Independently computed
- Global result
  - Computed combining sub results

The first (and only, for a while) kind of parallelism considered
Data parallelism

Fortran
  ◦ DO INDEPENDENT

OpenMP
  ◦ #pragma omp parallel for ...

Bird Meertens formalism (Bakcus’ FP)
  ◦ Map, $\alpha$
Stream parallelism

Multiple items appearing one after the other
  ◦ Independently processed
  ◦ To get final result (delivering one after the other)

Usually
  ◦ Infinite streams
  ◦ Packets from network, instructions from memory, images from video, …
Data parallel (sub kind(s))

Map

Reduce/Scan

Stencil
Stream parallelism

Pipeline

Farm

$\text{loopback}(\Delta)$
Speculative parallelism

Compute things possibly non necessary
  ◦ To save time

  ◦ If-then-else
    ◦ Compute then and else while you compute condition
    ◦ Upon condition result, abort non taken branch

  ◦ Need mechanisms to cancel/commit
Control parallelism

Loops
- Sequencing
- Unrolling
Activity graph

Nodes
- Activities computing a result out of some input data

Arcs
- Dependencies (data)
- \( N \rightarrow M \) iff \( N \) produces something consumed by \( M \)

Maximally parallel graphs \( \rightarrow \) feasible & efficient ones
Tools

I have a parallel machine
What can I use to program it in parallel?

:: Tools, tools, tools

Languages, libraries, compilers, etc.
Operating system (calls)

Fork/exec
- Processes

Pthread_create, join
- Threads

Semaphores, Shmem segs, Sockets, Pipes, RPC ...
- communications, Synchronization
Processes (fork/exec)

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```
Threads (create/join)

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread \#%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++) {
        printf("In main: creating thread \#%ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
            printf("ERROR: return code from pthread_create() is %d\n", rc);
        exit(-1);
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
Pipes and sockets

```
pipe(fd);
if(fork()) {
    // read pipe
    ...
} else {
    // write pipe
    ...
}
```
Locks, semaphores

Lock
- Spinlocks, active
- Dependencies on memory model

Mutex
- Signal/wait, passive

Semaphore
- P/V, passive
De facto standard (shmem)

OpenMP
- Regions
- Parallel for, tasks, single, ...

Pragma based
- Sequential equivalence

```cpp
int threads = 100;
int id = 100;
#pragma omp parallel
{
    threads = omp_get_num_threads();
    id = omp_get_thread_num();
    std::cout << "hello from thread: ", id << " out of ", threads;
}
return 0;

int i;
double xdoty;

xdoty = 0.0;

#pragma omp parallel \
    shared ( n, x, y ) \
    private ( i )
#pragma omp for reduction ( + : xdoty )
for ( i = 0; i < n; i++ )
{
    xdoty = xdoty + x[i] * y[i];
}
return xdoty;
```
De facto standard (distributed)

**MPI**

- Hundreds of primitives
  - Collectives, point-to-point
- SPMD model
- One side communications
- Hierarchical communicators

```c
int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor \%s, rank \%d\" 
           " out of \%d processors\n", 
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```
Parallel programming languages

A huge number

- A few actually used outside the designer *entourage*
- Providing some of the features needed
- Very good for a few kind of parallel applications
Sample parallel language (functional)

Erlang

- From mobile phone industry (Ericsson)
- Functional language, untyped

- Primitive mechanisms
  - To fork a computation
  - Send and receive from a forked thread

```erlang
do(Fun,X,A) ->
    Y = Fun(X), % apply(ff,Fun,[X]),
    A ! {self(),Y}.
reify(Pid) ->
    receive
    {Pid,Y} -> Y
    end.
```

spawn(Module, Function, Args) -> pid()
Sample parallel functions (imperative)

Cilk
- Spawn/wait mechanism (primitive)
- Parallel loop(s) added
- Array notation

```cilk
y[0:n] += alpha * x[0:n];
```

```cilk
// Compute $\sum$ foo(i) for i from 0 to N, in parallel.
cilk::reducer_opadd<float> result(0);
cilk_for (int i = 0; i < N; i++)
    result += foo(i);
```

```cilk
1. cilk int fib (int n)
2. {
3.     if (n < 2) return n;
4.     else
5.         {
6.             int x, y;
7.             x = spawn fib (n-1);
8.             y = spawn fib (n-2);
9.             x + y;
10.         }
11.     }
12. }
```

Sample parallel language (OO)

X10 is
- A language
  - Scala-like syntax
  - object-oriented, imperative, strongly typed, garbage collected
  - focus on scale ➔ focus on parallelism and distribution
  - focus on productivity
- An implementation of the APGAS programming model
  - Asynchronous Partitioned Global Address Space
    - PGAS: single address space but with internal structure (➔ locality control)
    - asynchronous: task-based parallelism, active-message-based distribution
- A tool chain
  - compiler, runtime, standard library, IDE
  - started as open-source research prototype, currently used for production
APGAS in X10: Places and Tasks

- Local Heap
- Global Reference
- Activities
- Place 0
- Place N

Task parallelism
- `async S`
- `finish S`

Place-shifting operations
- `at(p) S`
- `at(p) e`

Concurrency control within a place
- `when(c) S`
- `atomic S`

Distributed heap
- `GlobalRef[T]`
- `PlaceLocalHandle[T]`

APGAS Idioms

- Remote evaluation
  \[ v = \text{at}(p) \text{evalThere}(|arg1, arg2|); \]

- Active message
  \[ \text{at}(p) \text{async runThere}(|arg1, arg2|); \]

- Recursive parallel decomposition
  ```scala
  def fb(n: Long): Long = { 
    if (n < 2) return n;
    val f1: Long;
    val f2: Long;
    finish {
      async f1 = fb(n-1);
      f2 = fb(n-2);
      }
    return f1 + f2;
  }
  ```

- SPMD
  ```scala
  finish for(p in places()) { 
    at(p) \text{async runEverywhere}(); 
  }
  ```

- Atomic remote update
  ```scala
  at(ref) \text{async atomic ref}() += v;
  ```

- Data exchange
  ```scala
  // swap l() local and r() remote
  val _l = l();
  finish at(r) async { 
    val _r = r();
    r() = _l;
    at(l) async l() = _r;
  }
  ```