Everyday Parallelism

Robert Strzodka
Integrative Scientific Computing
Max Planck Institut Informatik
www.mpii.de/~strzodka
Parallelism

- Sequential execution is an illusionary software concept

- **All transistors always do something in parallel!**

- **Billions of transistors** in modern CPUs (>0.5) & GPUs (>2)

- **Old:** Implicit parallelism with caches, ILP, speculation → diminishing returns, power constraints

- **New:** Explicit parallelism on multiple levels → much more efficient & natural
SIMD Parallelism

- It is impossible to execute just one instruction
  - \( a = b + c; \)
  - Actually means execute \((\text{add, nop, nop, nop, ...})\)

- Penalty for ignoring SIMD
  - 4x on current CPUs (SSE)
  - 8-16x on future CPUs (AVX, LRB)
  - 16-80x on GPUs
Many-Core Parallelism

- **Penalty for ignoring many-cores**
  - 4-12x on current CPUs
  - 32-48x on future CPUs (LRB)
  - 10-30x on GPUs
Intra-Node Parallelism (multiple CPUs/GPUs per PC)

- Penalty for ignoring intra-node parallelism
  - 4-8x for multi-CPU systems
  - 4-8x for multi-GPU systems
Inter-Node Parallelism in a Cluster

- **Penalty for ignoring inter-node parallelism**
  - Depends on the number of nodes in the cluster
  - Capability computing
Overview

• Levels of Parallelism

• **Work Distribution**

• Specialization & Synchronization

• Data Movement

• Parallel Programming
Work Distribution inside a Company

- Chip
- Groups of Transistors
- Software
- Programmer

- Company
- Workers
- Tasks
- Boss
Boss with Parallel Work Distribution

- **Company setup**
  - Analyze work task distribution
    - A 60%, B 30%, C 5%, …
  - Hire workers accordingly

- **Execution setup**
  - Analyze static and possible dynamic task dependencies
  - Identify best work configurations for tasks

- **Execution**
  - Send input data and dynamic parameters

- **Mix of static and dynamic scheduling**
Boss with Old School Philosophy

• **Company setup**
  – No analysis of task distribution
  – Hire expensive generalists

• **Execution setup**
  – Compile all tasks into a big pile, do not encode task dependencies
  – Do not analyze or tell anyone what is in the pile

• **Execution**
  – Throw the data and the pile of tasks at your workers and yell

I WANT RESULTS NOW!
Boss with Parallel Work Distribution

- **Company setup**
  - Analyze work task distribution
    - A 60%, B 30%, C 5%, ...
  - Hire workers accordingly

- **Execution setup**
  - Analyze static and possible dynamic task dependencies
  - Identify best work configurations for tasks

- **Execution**
  - Send data and dynamic parameters

- **Mix of static and dynamic scheduling**
Overview

• Levels of Parallelism

• Work Distribution

• **Specialization & Synchronization**

• Data Movement

• Parallel Programming
Specialization – Granularity

Example: SIMD width

- **Hardware**
  - Coarser is better, because more resources are shared

- **Software**
  - Finer is better, because of better resource utilization
  - Abstraction: kernel functions (SIMT)

- **HW/SW Challenge**
  - Optimization problem: Coarse benefits common case, fine has less overhead on divergent branches
Specialization – Heterogeneity

Example: single and double precision FPUs

- **Hardware**
  - Higher is better, because specialized resources are more efficient

- **Software**
  - Lower is better, because scheduling is easier
  - Abstraction: Compiler on fine, libraries on coarse level

- **HW Challenge**
  - Optimization problem: best configuration for assumed work load distribution. Even more difficult with reconfigurability.
Mixed-Precision GPU-Multigrid Performance

- **Core2Duo G80-CPU GT200-CPU**
- **GT200-db GT200-GPU**

![Graph showing performance comparison between different hardware configurations.](image)

- Smaller is better

IMA Jan 2010
Synchronization – On-Chip

- **Hardware**
  - Bus, network on chip (NoC), shared memory, active research area

- **Software**
  - Currently: same synchronization on all levels
  - Abstraction: work queues, atomics, kernel barriers

- **HW/SW Challenge**
  - At some stage hierarchical synchronization will be necessary in HW
  - Synchronization of subgroups, lock-free data structures
Adaptive Representation of a Function
Enforcement of One-Level Transitions
Synchronization – Off-Chip

Multi-CPU system

Multi-GPU system

Cluster
Bandwidth in a CPU-GPU System

- CPU
  - processing elements
  - cache 40 GB/s
- co-processor
- system memory 6-30 GB/s
- device memory 20-160 GB/s
- Infiniband to next node 1-2 GB/s

IMA Jan 2010
Scalable Recursive Clustering (ScaRC)

- **A tradeoff between global coupling and locality**
  - Run a global MG (coupling of data on all scales)
  - With a block-Jacobi smoother (local computation)

- **Typical 2-level ScaRC in FEAST**
  - Global BiCGStab, preconditioned by
  - Global multigrid (V-cycle, 1+1 steps), smoothed by
    - Block-Jacobi, block inverses computed with a
  - Local multigrid (V/F-cycle, 2+2 steps), smoothed by
    - Local Jacobi / ADI-TRIDI / ADI-TRIGS

Joint work with Dominik Göddeke, TU Dortmund
Poster: Mixed-Precision GPU-Multigrid Solvers with Strong Smoothers
Overview

• Levels of Parallelism
• Work Distribution
• Specialization & Synchronization
• Data Movement
• Parallel Programming
Data Access

• **It is impossible to read one byte from memory**
  – `char one = array[5];`
  – This will typically fetch **128 bytes**
  – DRAM burst: |****open page****|****transmit data****|

• **Latency hiding**
  – Overlay computation and communication (double buffering)
  – Abstraction: Massive multi-threading (on-chip register-files)
  – Linear access still important

• **Bandwidth wall**
  – Quad-core Xeon X5482 3.2GHz
  – Measured peak 6.2 GB/s, 40.8 DP-GFLOPS
  – Balanced arithmetic intensity for doubles: 52.6
Company: Transport Logistics
Old School: Transport Logistics
Old School Transport Logistics Hitting the Bandwidth Wall

- for(int t = 0; t<iterNum; t++) {
  for(int ya = 0; ya<ySize; ya++) {
    for(int xa = 0; xa<xSize; xa++) {
      stencil_computation(t, xa, ya);
    }
  }
}

- If ySize*xSize*sizeof(MyDataType) < cacheSize then everything runs in cache

- Otherwise, repeated cache thrashing
  - Bandwidth wall is upper bound for performance
  - Cache blocking does not change this
Cache Oblivious Stencil Computations Xeon X5482

25.1 GFLOPS

19.1 G = 76% of peak

Naive 1.9 G  PluTo 8.2 G

11282^2 ≈ 1 GiB

Strzodka et al. Cache oblivious parallelograms in iterative stencil computations. ICS 2010

IMA Jan 2010
Discretization Grids
Structured and Unstructured Sparse MatVec

Larger is better

chart from [Bell and Garland SC 2009]
Deformation Adaptivity

- This grid is a generalized tensor-product

- Pro: Banded matrix achieves highest bandwidth

- Con: Highly anisotropic elements require strong solver (poster)
Overview

- Levels of Parallelism
- Work Distribution
- Specialization & Synchronization
- Data Movement
- Parallel Programming
Abstractions

- **Granularity**: kernel functions
  
  **Heterogeneity**: Compiler on fine, libraries on coarse level

- **Synchronization on-chip**: work queues, atomics
  
  **Synchronization off-chip**: vertical interfaces

- **Latency**: massive multi-threading
  
  **Memory hierarchy**: cache oblivious algorithms

- **Specialization**: domain specific languages
  
  **Optimization**: support for efficient programming patterns
Programming: Tunable Parameters

- {  
  \texttt{tunable} int Bx, By;
  for(int ya= 0; ya<ySize; ya+= By) {
    for(int xa= 0; xa<xSize; xa+= Bx) {
      block\_computation(xa, ya, Bx, By);
    }
  }
}

- Close integration of auto-tuning into compilers

- Guided, interactive optimization process
  - \texttt{tunable} int Bx(2, 10);
Programming: Execution Order 1

• 
  ```
  {  
    funcA();  
    funcB();  
    funcC();  
  }
  ```

• **This unnecessarily** defines the **execution order** even if A, B and C are independent

• `funcA() : in(a,b), out(c) { ... }`
  `funcB() : in(d), inout(e,f) { ... }`
  `funcC() : in(c), out(g,h) { ... }`

• **Dependencies** should determine the execution order
• for(int t= 0; t<iterNum; t++) {
  for(int ya= 0; ya<ySize; ya++) { for(int xa= 0; xa<xSize; xa++) {
      stencil_computation(t, xa, ya);
    }
  }
}

• Compiler warning: deprecated programming pattern
  – Do you really want to execute 10x slower?

• This unnecessarily defines the traversal order of the index space

• foreach( index-space; dependencies ) { … }
Programming: Data Layout (AoS vs. SoA)

Array of Structs (AoS)

```c
struct NormalStruct {
    Type1 comp1;
    Type2 comp2;
    Type3 comp3;
};

typedef NormalStruct AoSContainer[SIZE];

AoSContainer container;

container[5].comp3++;
```

Struct of Arrays (SoA)

```c
struct SoAContainer {
    Type1 comp1[SIZE];
    Type2 comp2[SIZE];
    Type3 comp3[SIZE];
};

SoAContainer container;

container.comp3[5]++;```

IMA Jan 2010
Summary

- **All transistors always do something in parallel!**
  - Let them do something meaningful
  - Sequential processing is an illusion even on single-core CPUs

- **Successful parallel work distribution and transportation**
  - Many companies do it every day
  - Why not scientists too?

- **Efficient parallel programming patterns are known**
  - But little used in practice
  - Even simple abstractions have not been adapted in languages

- **Education & Integration**
  - Dissemination of knowledge and best practices across disciplines
Questions?

Robert Strzodka
Integrative Scientific Computing
Max Planck Institut Informatik
www.mpii.de/~strzodka