Overview

• Structure of a parallel computer
• Parallel Software for 16 cores (CPU)
• Parallel Software for 1,600 cores (GPU)
Programming Parallel Systems

- So far, we talked (mainly) about storage systems
  - Main question: How can we guarantee a consistent system state

- Already desktop systems can be used for parallel computation
  - Distribute work load in the system!
  - How can we do this?
  - What’s underneath the hood?
Programming Parallel Systems: Basic Idea

• Our model for parallel programming:

• A job is split into many small tasks
  – These tasks can be executed in parallel

• The tasks can be distributed
  – Each „worker thread“ may get many tasks

• The partial results may be merged
  – This is just another kind of “task”
Programming Parallel Systems: Promise and Reality

**Promise**

- Memory

**A Real Computer?**

- Processor 4
  - Memory/4
- Processor 3
  - Memory/4
- Processor 1
  - Memory/4
- Processor 2
  - Memory/4

- Fast
- Not so fast
Programming Parallel Systems: Promise and Reality

Promise

A Real Processor

Fast

Memory

Proc. Core

Cache

Proc. Core

Cache

Memory/4

More Processors ...

... with more Memory

Not so fast

Slow
• Need to know your hardware for maximum efficiency
  – Cache Sizes, Topology & Bandwidth of Buses
  – Think: Data locality, (hidden!) communication cost
Programming Parallel Systems: A To-Do List

• We need to
  – write code for worker threads
  – distribute the threads to the cores
  – split the job into smaller tasks (how small?)
  – assign tasks to threads
  – balance the load on all threads
  – collect the (partial) results from the machines
  – assembly the results

• Should be fast as well, i.e., make use of locality
  – cache locality and prefer local memory over remote memory!

• The complexity of the program increases significantly!!!

• Solution?
OpenMP

- OpenMP is a specification developed by AMD, Cray, IBM, Intel, NVIDIA, ...
  - Parallelization
  - Load balancing
  - Implicit use of locality
    - If you know what you are doing
  - All in one library!

- Not really a library, but a language-extension
  - C, C++, Fortran (still used in scientific computing)

- Supports Basic Parallel Constructs
  - Loops, basic reductions, tasks, ...
  - Synchronization
OpenMP: An Example

```cpp
std::vector<int> a(N);
std::vector<int> b(N);

void sequential()
{
    for (int i = 0; i < N; i++)
    {
        a[i] *= b[i];
    }
}

void parallel()
{
    #pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        a[i] *= b[i];
    }
}
```

- Split loop into tasks
- Distribute tasks to workers

---

4 Procs x 4 Cores = 16 Threads
Speedup: 3.1x

Only 3.1x?
void parallel()
{
    #pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] *= b[i];
}

Iteration of for-loop:

0.. K.. 2K.. ...

N-K..N

Where are the memory cells accessed in iteration i?

When a worker is free: grab the next available task (block of iterations)
OpenMP: Digging Deeper

- Physical memory location depends on **Operating System**
  ```cpp
  std::vector<int> a(N);
  std::vector<int> b(N);
  ```

- **Virtual** Memory presented as continuous block
  - **Physical** Memory may be scattered
  - A single **page** of virtual/physical memory cannot be scattered
  - Typical page sizes: 4KB, **SuperPage**: 4MB

- Many OSes
  - Explicitly: Offer system call to pin a page to a physical processor by hand
  - Implicitly: Pin virtual pages to processor that first accesses it
  - How is this done?
OpenMP: Static Scheduling

```c
int *a, *b;
a = (int*)malloc(N*sizeof(int));
b = (int*)malloc(N*sizeof(int));

void fill()
{
#pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) {
        a[i] = a_value(i);
        b[i] = b_value(i);
    }
}

void parallel()
{
#pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i)
        a[i] *= b[i];
}
```

Chunk x is assigned to thread x mod num_threads

4 Procs x 4 Cores = 16 Threads
Speedup: 6.7x

Only 6.7x?
Not Every Parallel Program is a for-loop

- Barely scratched the surface of OpenMP
  - Reductions

```c
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += a[i] + b[i];
}
```
Not Every Parallel Program is a for-loop

- Barely scratched the surface of OpenMP
  - Reductions
  - Arbitrary task types

```cpp
# ifndef A
# define A
A
```

```cpp
// or

A car race
A race car
(or)
A car
(car race
(or...)
```
Not Every Parallel Program is a for-loop

- Barely scratched the surface of OpenMP
  - Reductions
  - Arbitrary task types
  - Synchronization primitives

```cpp
cout<<"A ";
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      cout<<"car ";
    }
    #pragma omp task
    {
      cout<<"race ";
    }
    #pragma omp taskwait
    cout<<"is fun to watch";
  }
}
cout<<endl;
```

A car race is fun to watch
(or)
A race car is fun to watch
Not Every Parallel Program is a for-loop

• Barely scratched the surface of OpenMP
  – Reductions
  – Arbitrary task types
  – Synchronization primitives
  – ...

• Already a simple loop can be tricky

• Simple loops are everywhere!
  – Think: Vectors, Matrix Multiplication
  – Simple loops deserve their own hardware
Graphic Processing Unit (GPU)

- The complexity of the architecture increases further

- The GPU consists of compute units, each with multiple stream cores
  - As an example, AMD Radeon R9 290X has 2816 stream cores
The Real Deal

Programming Guide
AMD Accelerated Parallel Processing
OpenCL™

[Diagram of Compute Unit <-> Memory Channel Xbar with multiple Compute Units and Memory Channels, with labels for L1, L2, WC, Complete Path Atomics, and Channel Address calculations: ((Address / 256) % n) == 0, 1, n-2, n-1]
Graphic Processing Unit (GPU)

- Different compute units can do different things
- All stream cores execute the same instruction sequence
  - With separate local memories

```
  Compute unit
  \hspace{1cm}
  Compute unit

  Stream cores
  \hspace{1cm}
  Stream cores

A + B \hspace{1cm} A + B
\hspace{1cm} A + B

A + B \hspace{1cm} A + B
\hspace{1cm} A + B

D \ast C \hspace{1cm} D \ast C
\hspace{1cm} D \ast C

D \ast C \hspace{1cm} D \ast C
\hspace{1cm} D \ast C
```

- What is this good for?
Matrix Operations

- Matrix operations are the core of graphics computations

- For example, matrix multiplication can be highly parallelized

- Naive: $O(n^3)$ multiplications
Matrix Multiplication

• Naive:  
  \( O(n^3) \) multiplications  
  – Small rounding errors

• Better: Strassen  
  \( O(n^{2.807}) \) multiplications  
  – Re-use partial results  
  – Can also be done in parallel

• Even better? Coppersmith-Winograd  
  \( O(n^{2.375477}) \) multiplications  
  – Asymptotically better  
  – But not for practical matrix sizes
All-Pairs Shortest Path

- Some problems can be represented nicely by matrices

- Let \( G = (V, E) \) be a connected graph. The adjacency matrix \( M \) of \( G \) has a 1-entry on \( M(u, v) \) if there is an edge between nodes \( u \) and \( v \)

\[
\begin{array}{ccccccccccc}
A & B & C & D & E & F & G & H & I & J \\
A & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
B & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
C & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
D & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
E & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
F & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
G & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
H & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
I & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
J & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]
All-Pairs Shortest Path

- The adjacency matrix gives us all nodes at distance 1

- To get nodes at distance 2, multiply the adjacency matrix by itself

\[
M^2 (A, F) = M(A, A)M(A, F) + M(A, B)M(B, F) + \ldots + M(A, F)M(F, F) \geq 1
\]
Solving the All-Pairs Shortest Path Problem

- Similarly, get nodes at distance 3 by multiplying $M^2$ by $M$:
  \[ M^3(A, I) = M^2(A, A)M(A, I) + M^2(A, F)M(F, I) + ... \geq 1 \]
All-Pairs Shortest Path

• After \(i\) multiplications, \(M(u, v) \neq 0\) if there is a path of length at most \(i + 1\) from \(u\) to \(v\)

• After \(\text{diameter}(G) - 1\) multiplications, we have found all nodes

• The length of the shortest path between any two nodes \(u\) and \(v\) is the index of the step \(i\) for which, \(M^{(i-1)}(u, v) = 0\) and \(M^i(u, v) \geq 1\)
  – Write distances to output matrix \(Q\)

• We can store the partial paths found in the intermediate steps
  – get the actual shortest paths in the end
Conclusion

• OpenMP
  – Widely used in scientific computing
  – CPUs execute ‘real’ threads
    – Don’t have to execute the same line of code everywhere

• GPUs have way more cores than CPUs
  – Enables more parallelism
  – Cores execute the same instruction per clock cycle
  – Efficient for matrix operations
  – Can be programmed using
    – OpenCL
    – CUDA
    – possibly OpenMP in the future
Outlook

• Faults
  – OpenMP, OpenCL, CUDA don’t care about faults
  – Hadoop/MapReduce: Store all intermediate steps, for fault-tolerance
  – Apache Spark: Recompute intermediate steps in case of (rare) faults

• Bottlenecks
  – Solution to problem designed around the shortcomings of the hardware
  – Why don’t we design the hardware around our problem?
    Remove bottlenecks, fine-tune relative speed of system components
  – «MinuteSort with Flat Datacenter Storage», MSR
    Disk reads can be a bottleneck as well → Design whole datacenter around it
    Overlap disk reads with asynchronous sorting-passes of already available data
    Unbeaten entry from 2012 for ‘Number of elements sorted in 60 seconds’
    www.sortbenchmark.org
That’s all, folks!

Questions & Comments?

Roger Wattenhofer