A24: DAY 1
PART2: APPLICATIONS
Parallel Programming

- Choose/design algorithm
- Parallelize algorithm
  - Expose enough layers of parallelism
  - Minimize communication, synchronization, dependencies
  - Overlap computation and communication
- Implement parallel algorithm
  - Choose parallel programming model
  - (?) Choose platform
- Tune/optimize application
  - Understand performance bottlenecks & expectations
  - Apply platform specific optimizations
  - (?) Apply application & data specific optimizations
BUILDING PARALLEL APPLICATIONS
From Application to Machine

- Specify an application as a model of concurrent computation
- Refine the model
  - Design: algorithm
  - Implementation: code
  - System: execution environment
  - Machine: architecture, real system
Example

• Specification level (the problem)
  • Sum n numbers
• Design level (idea of the algorithm):
  • 1 number per processing unit
  • Read sum, Add my number, Write new sum
• Implementation
  • Include programming-model agnostic restrictions:
    • Avoid race condition for read/write the sum
• System (programming model)
  • Use PM constructs to write code
• Machine
  • Adds mapping restrictions
Simplified Parallel Application Design

- Decompose
  - What is the computation? Which data?
- Cluster
  - What granularity?
- Implement in programming model
  - Which model?
  - Implement tasks as processes or threads
  - Add communication and synchronization constructs
- Map units of work to processors
  - Might be transparent.
Decompose

- Goal: minimize data dependencies between concurrent units
  - Decide what level of parallelism to use
  - Decide which data (dimension) you will use
  - Static or dynamic?
  - Level of concurrency should depend on problem size
    - What is problem size
Cluster

- Goal: identify the right granularity
  - Bring tasks (and sometimes data) together in conceptually executable programs (processes or threads)
  - Various mechanisms
    - cluster iterations
    - data driven decomposition
    - use heuristic cluster algorithms
Implement in programming model

• Choose programming model
  • Add communication and synchronization constructs
  • Add access mechanisms to shared data
  • Design distributed data structures
• Optimize program to reduce cost of parallelism
  • combine communication messages
  • remove redundant synchronizations
  • reduce parallel loop overhead
Map work to processors

- Find the right way to make use of computation resources
  - Assign processes or threads (tasks) to processors
    - Might be transparent!
  - Optimize mapping
    - For locality: tasks that communicate a lot on the same processor
    - For load-balancing: distribute work of similar complexity
- Mapping can be static or dynamic
  - Static allocation: affinity
  - Dynamic allocation: OS and/or RTS
Models of Parallel Computation

• Allow application classification
  • Implementation follows this classification

• Enable more parallelism in the application
  • See Farmer-worker

• Enable structure in the application
  • See divide-and-conquer

• Improve load balancing

• Allow reasoning about complexity and performance
Models of Parallel Computation

• Conceptual level
  • Trivially parallel
  • Master/worker
    • Pull vs. push
  • Divide and conquer
  • Data parallelism
  • Task parallelism
    • Pipelining, data-flow, …
  • Bulk-synchronous

• System level: Shared memory vs. distributed memory
  • (logical) data spaces and programs
  • communication and synchronization
Trivially parallel

- Independent computations for each data element
  - Data-driven
  - Decompose always at the finest granularity
Master/worker model

- A pool of identical *worker* processes is formed,
- The *farmer* manages the work for the workers
- When finished, a worker gets another job until nothing is left.
  - **Pull model**: workers ask for work
  - **Push model**: farmer gives work to workers
Divide and Conquer

- A task is recursively split in smaller tasks
  - Fine grained tasks (i.e., leaves) are handed to workers
  - Results are recursively aggregated towards the root
  - Natural concurrency, as different subproblems can be solved concurrently
Data parallelism

- Data driven parallelization:
  - data is distributed to the memories of the processors.
- Computation follows the data

interconnection network

data structure

block distribution

processor-memory pairs
Task parallelism

- Distributes concurrent tasks to parallel computing nodes.
- Subclasses (by execution model):
  - pipelining
    - Different task per node
    - The intermediate results of the current task/node are used by the following node
  - data-flow (data-driven) execution
    - A task is executed as soon as all its input arguments are available
  - demand-driven execution
    - A task is only executed if its output is needed as input for another task
Bulk synchronous (BSP)

- Parallel computation consists of repeated cycles of
  - Compute phase: all calculations are done locally
  - Communication phase (+ global synchronization)
  - Decision Phase (e.g. convergence)
  - Communication Phase (+ global synchronization)
Exercise (teams of 2-3)

• Imagine an application that can be parallelized using:
  • Master-worker
  • Data parallelism
  • Task parallelism
  • Divide and conquer
  • Bulk synchronous

• Specify
  • Application scenario
  • Parallelism model
  • Expected performance gain
System level: Communication

Shared memory
- Accessible data
- R/W ordering problems
- **Races** => non-determinism

Distributed memory
- Message passing for accessing and modifying data
- Consistency problems => non-determinism
Communication in shared memory

Mutual exclusion is required

thread 1

// thread-1
...
lock s;
s = s+4;
unlock s;
...

thread 2

// thread-2
...
lock s;
s = s+5;
unlock s;
...

in global memory

S = ?
Synchronization in shared memory

- Some parallel models, like the bulk synchronous model, require global synchronization.
- Barrier construct: `barrier(name, processor_group)`

```c
// code of thread-1
...
// compute code
barrier(5, all_procs);
// communication or
// updating shared data
barrier(6, odd_procs);
// decision code
barrier(7, even_procs);
```

```c
// code of thread-2
...
// compute code
barrier(5, all_procs);
// communication or
// updating shared data
barrier(6, odd_procs);
// decision code
barrier(7, even_procs);
```
Communication in message passing

- Basic building blocks (not all parameters are shown):
  - `send(void *sendbuf, int n_elems, int id_dest)`
  - `receive(void *recbuf, int n_elems, int id_source)`

Both receive and send can be
  - Blocking/non-blocking
  - From/to any other process

// code P0
...
  a=100;
  send(&a, P1);
  a=0;
  ...
// code P1
...
  receive(&a, P0);
  printf("%d\n", a);
  ...

Synchronization in message passing

// code P0
...
send(&a, P1);
receive(&a, P1);
a++;

// code P1
...
send(&a, P0);
receive(&a, P0);
a++;

What happens is send is blocking?

What happens is receive is blocking?

What happens if send is blocking and receive is not blocking?
Entry points for parallelization

• From sequential code base:
  • Profiling => identify computationally intensive kernels
  • Code inspection => identify computationally intensive loops
  • Data structures inspection => unbundle them
  • Code rewriting using parallel constructs or library calls

• From algorithm(s):
  • Some sequential algorithms cannot be parallelized
  • Use a suitable parallel algorithm

• From scratch (see previous slides):
  • Working with existing code base is sometimes inefficient
  • Design parallel algorithm from specification
Practical exercises [part 1]

- Histogram of a $N \times N$ image into $B$ bins
  - Can consider gray-scale, $B=256$
  - Can choose anything else

- Apply a convolution filter $k \times k$ to a $N \times N$ image
  - Can choose Sobel filter, but don’t have to

- Parallelize the All Pairs Shortest Path in a graph given as a list of edges

- Caesar’s code
  - Given text file
  - $N$-value Key
Answer

- Parallelization process
  - What is your parallel algorithm?
  - Did you have multiple choices?
  - Which one have you chosen and why?

- Build an experimental setup
  - Best case/worst case? Why?
  - Best machine/worst machine? Why?

- What can you say about the performance?
  - Performance bottlenecks?
  - Any other performance considerations?
PARALLEL PROGRAMMING MODELS
Parallel Programming Models

- Programming model = language + libraries to create a model of computation

- Abstraction level = qualifier for the distance between the abstract and the real machine

  - Classification:
    - High-level languages:
      - Pro: Programmer friendly
      - Con: Performance impacted by overheads
    - Low-level languages:
      - Pro: Performance
      - Con: Difficult to learn and use
Classification attempts

• Control model
  • Parallel operations creation
  • (Logical) Ordering

• Data model
  • Shared vs. Private data
  • Logical access to data (i.e., communication)

• Synchronization model
  • Coordinating parallelism

• Cost model(/performance impact)
  • Expensive vs. Cheap operations
  • Overhead
Examples

- Classical models
  - Shared memory
    - OpenMP
    - Pthreads
  - Distributed memory
    - MPI
    - HPF
- Alternatives
  - Intel TBB, ArBB
  - Cilk/Satin
  - UPC (PGAS languages)
  - SAC
- Models/languages for heterogeneous architectures
  - OpenCL
  - OpenACC
COMPARING PROGRAMMING MODELS
Criteria

• Programmability =
  • Performance + Productivity + Portability
• Performance
  • Preserve the platform performance.
• Productivity
  • Allow for quick/clean implementation, close to the application
• Portability
  • Code: use the same code
  • Parallelism: use the same parallel model
  • Performance: preserve the level of performance
Metrics

- **Performance**
  - Quantitative: Execution time, speed-up, efficiency

- **Productivity**
  - Qualitative: User-friendly-ness, level of abstraction
  - Quantitative: Lines of Code, #bugs/line of code

- **Code portability**
  - Qualitative: Y/N
  - Quantitative: # required changes

- **Parallelism portability**
  - Qualitative: Runs in parallel?

- **Performance portability**
  - Quantitative: how well does it perform?
In real life ...

- Challenges in quantitative comparisons of programming models:
  - Obtain similar implementations?
    - Different borders between front-end and back-end
    - Different transparent implementations
  - Test in similar conditions?
    - Different compilers
    - Different programmers

- Comparing programming models remains an open question.
  - Adoption might be correlated with the ease-of-use and performance, but it all seems to be a matter of personal preference.
Open question

- Application => PM?
- Application => Platform?
- Platform => PM?
- PM => platform?

- Choosing the right pairing is an open question
  - Solutions
    - Application characterization (TUDelft)
    - Dwarves (UC Berkeley)

Bottom line: in most cases, these choices remain arbitrary.