Designing parallel applications
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.
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
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
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
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
A task is recursively split into 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
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
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)
System level: Communication

Shared memory
- Accessible data
- R/W ordering problems
- Non-determinism?

\[ d = e + a; \]

\[ a = 5; \]
in global memory

Distributed memory
- message passing for accessing and modifying data
- Consistency problems
- Non-determinism?

\[ a \]
in local memory
Communication in shared memory

Mutual exclusion is required

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

// thread-2
...  
lock s;
    s = s+5;
unlock 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);
```
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, 1, P1);
a=0;
... 

// code P1
... 
receive(&a, 1, P0);
printf(“%d\n", a);
...
Synchronization in message passing

// code P0
...
send(&a, 1, P1);
receive(&b, 1, P1);
...

// code P1
...
send(&a, 1, P0);
receive(&b, 1, P0);
...

Send is blocking or non-blocking?  Non-blocking
Receive is blocking or non-blocking?  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
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
OpenMP: Summary

- Shared memory programming model
- Explicit parallelism
  - Fork–join model
- Implicit data distribution
- Allows for data and task parallelism
- High abstraction level
- Preserves sequential code (when possible/needed)
pThreads: Summary

- Thread management is explicit
  - Creation, joining, killing
- Thread scheduling is (typically) done by OS
  - No pre-determined model
- Communication
  - Explicit via shared variables
  - Protection for race conditions needed
  - Protection against deadlocks
- Synchronization
  - Locks
  - Barriers (not available in the standard!)
HPF: Summary

- Data-parallel programming model
  - Computation follows data
  - Communication and synchronization “by request”

- Data distribution & alignment
  - Essential for performance, locality, low communication/synchronization

- Different processors view
  - Allow for easier data distributions

- High abstraction level

- Performance depends on compiler
  - Large overheads depending on the machine
Message Passing: Summary

- No notion of global data
- Data communication is done by message passing
  - Expensive, performance-wise
- Trade-off between:
  - One-copy data
  - More communication is needed, less consistency issues
  - Local data replication
  - (apparently) Less communication, consistency is problematic
- Techniques to improve performance:
  - Replicate read-only data
  - Computation and communication overlapping
  - Message aggregation
Levels of abstraction

- Low level of abstraction:
  - Pthreads: shared memory
  - MPI/PVM (message passing): distributed memory
  - Very few limitations in expressivity
  - Express all parallel patterns

- Trend in parallel programming models = increase level of abstraction:
  - Increase in productivity
  - But might decrease performance
  - Decrease in expressivity
  - But target specific classes of problems

- Exception: hardware-centric models
Threading Building Blocks (Intel TBB)

- Library-based for C++
- High-level support for multi-threading
  - Application = collection of logical threads
  - Mapping logical to physical threads = transparent
- Focuses on computationally intensive work
- It targets data parallelism
  - But can support task-parallelism
- Generic programming
  - Object-oriented
Intel Array Building Blocks (ArBB)

- Library that targets data-parallelism
  - Simply put: independent iterations of loops over arrays
  - In ArBB terminology: containers, iterators, operators

- Three components
  - ArBB kernels: serial C++
  - Compiler
  - Run-time: includes scheduler and threading magic

- Allows for both vector and scalar processing

- Constructs:
  - Scalar and vector types and operators
  - Functions
  - Control flow (sequential!!)
Cilk (MIT, now at Intel)

- Targets logical multi-threading
  - Programmer writes logical threads
  - Scheduling is done automagically

- Based on C, designed for SMP’s
  - Very few added keywords
  - Most important: spawn and sync

- Dynamic multi-threading
  - The task-graph is generated “on-the-fly”
  - Typical for Divide-and-Conquer

- Greedy scheduling
  - Enables performance modeling and prediction

More information: supertech.csail.mit.edu/cilk
Cilk: Example

- Parallelize in Cilk
  - Convert loop to recursion
  - Insert Cilk keywords
- Recursion + keywords

```cilk
void VectorAdd(float *a, float *b, float *c, int n) {
  if (n < GrainSize) {
    int i;
    for (i = 0; i < n; ++i)
      a[i] = b[i] + c[i];
  }
  else {
    spawn VectorAdd(a, b, c, n/2);
    spawn VectorAdd(a+n/2, b+n/2, c+n/2, n/2);
    sync;
  }
}
```
Satin (Vrije Universiteit)

- Parallelism based on divide-and-conquer
  - Similar to Cilk
  - Based on Java, not C
- Designed for clusters and grids
  - Large scale, distributed machines
- Important distinction between divide-and-conquer parallelism and scheduling/mapping
- Allows for more advanced scheduling
  - Random work-stealing
    - Steal work from a random queue
    - Cluster-aware random work-stealing
    - Stay local to node, if possible
    - Master-workers
**UPC (George Washington University)**

- **UPC = Unified Parallel C**
  - Parallel specifications in C => requires compiler

- **Designed for the PGAS (Partitioned Global Address Space) model**
  - Single address space, with affinity
    - Can express/address locality
    - Synchronization/consistency issues

- **Similar languages:**
  - Titanium – UC Berkeley
  - Co–Array Fortran
UPC: more features

- Uses a SPMD model
  - A number of threads execute concurrently
  - Each thread has a piece of shared space + private space
- Provides control for data distribution
- Provides parallel constructs (e.g., upc_forall)
- Provides synchronization constructs
  - Barriers
    - Blocking barriers
    - Notify–wait (two–phase) barriers
  - Locks
Pattern-based languages: OPL

- Observation: most applications implementations are based on patterns
- In pattern-based programming:
  - Application = a collection of nested patterns
  - Application design = iterative refinement of patterns
- Pattern classification (high $\rightarrow$ low-level)
  - Structural patterns – i.e., the task graph
  - Computational patterns – i.e., the computation
  - Algorithmic patterns – i.e., concurrency
  - Implementation patterns – i.e., organizing program+data
  - Parallel execution patterns – i.e., low-level parallelism (HW)
<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Computational Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent and Repository</td>
<td>Pipe-and-filter</td>
</tr>
<tr>
<td>Arbitrary Static Task Graph</td>
<td>Process Control</td>
</tr>
<tr>
<td>Iterative_refinement</td>
<td>Backtrack Branch and Bound</td>
</tr>
<tr>
<td>Event-based, implicit invocation</td>
<td>Monte Carlo Methods</td>
</tr>
<tr>
<td>Layered systems</td>
<td>Circuits</td>
</tr>
<tr>
<td>Map reduce</td>
<td>N-Body Methods</td>
</tr>
<tr>
<td>Model-view controller</td>
<td>Dense Linear Algebra</td>
</tr>
<tr>
<td></td>
<td>Dynamic Programming</td>
</tr>
<tr>
<td></td>
<td>Spectral Methods</td>
</tr>
<tr>
<td></td>
<td>Structured Grids</td>
</tr>
<tr>
<td></td>
<td>Unstructured Grids</td>
</tr>
<tr>
<td></td>
<td>Sorting</td>
</tr>
<tr>
<td></td>
<td>Graph Algorithms</td>
</tr>
<tr>
<td></td>
<td>Graphical Models</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel Algorithm Strategy Patterns</th>
<th>Implementation Strategy Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Parallelism</td>
<td>SPMD</td>
</tr>
<tr>
<td>Recursive splitting</td>
<td>MasterWorker</td>
</tr>
<tr>
<td></td>
<td>sharedqueue.pdf</td>
</tr>
<tr>
<td>Discrete Event</td>
<td>LoopParallelism</td>
</tr>
<tr>
<td>Geometric Decomposition</td>
<td>Distributed Array</td>
</tr>
<tr>
<td></td>
<td>Shared Hash Table</td>
</tr>
<tr>
<td>Data Parallelism</td>
<td>shared data</td>
</tr>
<tr>
<td>Speculation</td>
<td>memory parallelism</td>
</tr>
<tr>
<td></td>
<td>(notes)</td>
</tr>
<tr>
<td></td>
<td>(Program Structure)</td>
</tr>
<tr>
<td></td>
<td>(Data Structure)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concurrent Execution Patterns</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MIMD</td>
<td>Thread Pool</td>
</tr>
<tr>
<td>Task Graph</td>
<td>Speculation</td>
</tr>
<tr>
<td>SIMD</td>
<td>Data Flow</td>
</tr>
<tr>
<td>DigitalCircuits</td>
<td>Message Passing</td>
</tr>
<tr>
<td></td>
<td>Collective Communication</td>
</tr>
<tr>
<td></td>
<td>Mutual Exclusion</td>
</tr>
<tr>
<td></td>
<td>P2P Sync</td>
</tr>
<tr>
<td></td>
<td>Collective Synchronization</td>
</tr>
<tr>
<td></td>
<td>Transactional Memory</td>
</tr>
<tr>
<td></td>
<td>(notes)</td>
</tr>
<tr>
<td></td>
<td>(Advancing Program Counters)</td>
</tr>
<tr>
<td></td>
<td>(Coordination)</td>
</tr>
</tbody>
</table>
OPL: Considerations

- Abstract, conceptual programming model
  - No clear implementation yet
- Performance depends on?
  - Identification of correct initial pattern
  - High performance back-end: implementation of the patterns for given computational infrastructure
- Portability?
- Productivity?

More information: http://parlab.eecs.berkeley.edu/wiki/patterns/patterns
Comparing programming models
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-friendliness, 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?
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.
## Interesting estimates*

<table>
<thead>
<tr>
<th>Model</th>
<th>Productivity</th>
<th>Portability</th>
<th>Performance</th>
<th>Programmability</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>--</td>
<td>--</td>
<td>+</td>
<td>--</td>
</tr>
<tr>
<td>MPI</td>
<td>--</td>
<td>+</td>
<td>+</td>
<td>--</td>
</tr>
<tr>
<td>OpenMP</td>
<td>+</td>
<td>~</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Cilk</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ArBB</td>
<td>+</td>
<td>~</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>TBB</td>
<td>~</td>
<td>+</td>
<td>+</td>
<td>--</td>
</tr>
<tr>
<td>CUDA</td>
<td>~</td>
<td>--</td>
<td>++</td>
<td>~</td>
</tr>
<tr>
<td>Brook+</td>
<td>+</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>PGI Accelerate</td>
<td>~</td>
<td>+</td>
<td>~</td>
<td>+</td>
</tr>
<tr>
<td>ENZO</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SDK</td>
<td>--</td>
<td>--</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>ALF</td>
<td>+</td>
<td>~</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>CellSs</td>
<td>++</td>
<td>+</td>
<td>~</td>
<td>+</td>
</tr>
<tr>
<td>SP@CE</td>
<td>+</td>
<td>~</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>OmPSS</td>
<td>++</td>
<td>+</td>
<td>~</td>
<td>++</td>
</tr>
<tr>
<td>Sequoia</td>
<td>~</td>
<td>+</td>
<td>~</td>
<td>+</td>
</tr>
<tr>
<td>Charm++</td>
<td>~</td>
<td>~</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>OPL (pattern-based)</td>
<td>+</td>
<td>++</td>
<td>?</td>
<td>++</td>
</tr>
<tr>
<td>OpenCL</td>
<td>~</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Open questions

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

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

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