Design Basics

Piotr Luszczek
Levels of Abstraction

**Concepts**

- Algorithms: partitioning, communication, agglomeration, mapping
- Messages, reductions
- Mutex, Semaphores, ...
- Atomics
- Memory coherency, transactions

**Tools**

- Domain, channel, task, locality, utilization
- MPI_Reduce(), #omp reduce:+
- lock()/unlock(), V()/P()
- compare_and_swap()
- vmovnrngoaps, clevict1
Example: Largest Value (sequential)

- C code

```c
for (int i=0; i<N; ++i) X[i] = rand()

int largest = X[0]

for (int i=1; i<N; ++i)
    if (largest < X[i])
        largest = X[i]

printf("%d\n", largest);
```

- Complexity:
  - O(N) (memory accesses, comparisons)
Example: Largest Value (threaded)

- Code
  ```c
  for (int i = 0; i < N; ++i) thread_create(thread_max, &X[i])
  void thread_max(int *X) {
    lock();
    if (largest < X[0])
      largest = X[0];
    unlock();
  }
  ```

- Complexity
  - \(O(1)\)\(*num\_threads\) (comparisons)
  - \(O(N)\) (memory accesses, locks)

- Amdahl fraction (sequential part)
  - 100%!!!

- Scaling (Gustafson)
  - Does not scale
Example: Largest Value (OpenMP)

- Code
  
  ```c
  #pragma omp parallel for reduction(max:largest)
  for (int i=0; i<N; ++i)
    if (largest < X[i])
      largest = X[i]
  ```

- Complexity
  - O(N)/num_threads (comparisons)
  - O(N/num_threads)*num_threads (memory accesses)

- Amdahl fraction
  - s → 0% as N → ∞
  - Beware of hidden cost of reduction

- Scaling (Gustafson)
  - Good...
    - As long as OpenMP reduction scales
Example: Largest Value (MPI)

- **Code**
  
  ```c
  MPI_Reduce(X, N / P, MPI_MAX);
  P = MPI::Comm::World.size;
  ```

- **Complexity**
  - \(O(N)\) messages (no matter the implementation)
  - \(O(N)\) comparisons
  - \(O(N/P + \log P)\) global time steps

- **Amdahl fraction**
  
  - \(s \rightarrow \log P\) if \(P >> N\)
    - if \(P\) is small then \(N/P\) dominates

- **Scaling (Gustafson)**
  
  - Good
    - As long as \(\text{MPI\_Reduce()}\) scales
Design Methodology for Parallel Algorithms

- Proposed by Ian Foster
  - Book title
    - “Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering”
  - Publisher: Addison-Wesley, Reading, MA, 1995
  - Details repeated in course textbook by Michael J. Quinn

- Principle:
  - Focus on the problem
    - Use the language of the problem, not the machine
  - Delay machine-dependent details and issues

- Design steps
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping
Partitioning

- Divide computation into “small” pieces
- Approaches
  - Data-centric
  - Computation-centric
- Decompositions
  - Domain decomposition
  - Functional decomposition
- Result
  - (primitive) Tasks
  - Data items
Partitioning: Guidelines

- Have an order of magnitude more tasks than processors
  - Required to enough parallelism and further adjustments
- Redundant computation/storage is minimized
  - Important for scaling problem size
- Primitive tasks are roughly the same size
  - Important for load balancing
- Number of tasks is a function of problem size
  - Important for scaling the hardware with problem size
- Optimizations to keep in mind
  - Partitions (dimensionality, size) correspond (roughly) to hardware
Communication

• Communication is only necessary because of parallelism
  – It doesn’t exist in sequential algorithms

• Determine communication patterns
  – Local (Small group of processes communicate)
  – Global (Most of the processes communicate)

• Tasks communicate through channels
  – Visualize your channels to see how many you need
  – Estimate the amount of communication in the channels

• Ideally:
  – Communication is balanced
  – Communication occurs between small number of tasks
  – Communication is performed in parallel
  – Computation is performed in parallel

  • Good: {compute(); send()} || {compute(); receive()}
  • Bad: {compute(); send()} || {receive(); compute()}
Agglomeration

• Primitive tasks are grouped (agglomerated) to achieve:
  - Better performance
    • Lower communication overhead (bandwidth)
    • Smaller number of messages (latency due to message startup)
  - Simpler code
• Guiding principle: maintain (or increase) locality
  - Locality minimizes or eliminates communication
• Example agglomeration targets
  - Data dimensions
    • Merge dimension(s) for example use 1D instead of 2D
  - Channels with excessive communication
Agglomeration Guidelines

- Increase *locality* as much as possible
- *Replicated computation* must be shorter than communication it replaced
- The partitioning must still scale
  - Tasks and their data are still small enough
- Agglomerated tasks are similar (for load balancing) in terms of:
  - Computation
  - Communication
- Number of agglomerated tasks is a function of the global problem size:
  - Tasks = f(size)
- Number of agglomerated tasks is small but as large as number of processors:
  - Tasks > Processors
- The existing sequential code can be used for agglomerated tasks (with minimal modifications)
Mapping

• Mapping assigns tasks to processors
  - This should optimize for the the hardware

• Increase processor utilization
  - Processors should run in parallel
  - Processors should compute for (roughly) the same amount of time between communication exchanges

• Decrease communication
  - If a channel is mapped to the same processor, the communication through that channel may be removed
  - Make communication local
    • Channels should connect close groups of processors

• Often, finding optimal mapping is NP-hard
  - Many mapping problems can be reduced to graph coloring
Mapping: Decision Tree

- The number of tasks is **static**
  - The communication pattern **structured**
    - Roughly *constant* computation time per task
      - Agglomerate to minimize communication
      - One task per processor
    - Computation time per task *varies* by region
      - Cyclically map tasks to processors to balance communication load
  - The communication pattern is **unstructured**
    - Use static load balancing
- The number of tasks is **dynamic**
  - *Frequent* communication between tasks
    - Use dynamic load balancing
  - Many *short-lived* tasks and no intertask communication
    - Use a runtime task-scheduling
Mapping: Checklist

• Consider both designs:
  - One task per processor
  - Multiple tasks per processor

• Consider both task-processor allocations:
  - Static
  - Dynamic

• For dynamic task-processor allocation:
  - Ensure task allocation/management is not a bottleneck

• For static task-processor allocation:
  - Have an order of magnitude more tasks than processors
Back to “Largest Value” Example