Review: Creating a Parallel Program

- Can be done by programmer, compiler, run-time system or OS
- Steps for creating parallel program
- Decomposition
- Assignment of tasks to processes
- Orchestration
- Mapping

Programming for Performance

- Partitioning
- Granularity
- Communication
- Caches and Their Effects
Where Do Programs Spend Time?

- **Sequential**
  - Performing real computation
  - Memory system stalls

- **Parallel**
  - Performing real computation
  - Stalled for local memory
  - Stalled for remote memory (communication)
  - Synchronizing (load imbalance and operations)
  - Overhead

- **Speedup** \( (p) = \frac{\text{time}(1)}{\text{time}(p)} \)
  - Amdahl’s Law (low concurrency limits speedup)
  - Superlinear speedup possible (how?)

Partitioning for Performance

- **Balance workload**
  - reduce time spent at synchronization

- **Reduce communication**

- **Reduce extra work**
  - determining and managing assignment

- **These are at odds with each other**
  - e.g. communication reduced by using one processor
Load Balance and Synch. Wait Time

• Basic load balance equation:

\[
\text{Speedup}_{\text{problem}} \leq \frac{\text{Sequential work}}{\text{max work on any processor}}
\]

– work includes not only computation but communication and data access
– work should also be done at the same time

• Real goal: reduce time spent at synchronization points
  – including implied one at end of program

Load Balance and Synch. Wait Time

• Identify concurrency
• Managing concurrency
  – static
  – dynamic
• Granularity of concurrency
• Serialization and synchronization costs
Data vs. Functional Parallelism

- **Data Parallelism**
  - same ops on different data items
  - leads to SPMD programming
- **Functional (control, task) Parallelism**
  - Also called control parallelism or task parallelism
  - example: task pipeline (series of producers and consumers)
- Hybrids are possible: pipeline of data parallel tasks
- **Impact on load balancing?**
- **Functional is more difficult**
  - relatively few functions
    - longer running tasks
  - often requires more software development
    - consider 2K processors, each running a different functional task

Managing Concurrency: Load Balance

- **Static**
  - Algorithmic mapping of tasks to processes
    - e.g. example solver
  - Requires little task management overhead
  - Better with predictable amounts of work
  - Can not adapt to runtime events and data variations
- **Dynamic**
  - Semi-Static: assignment for a phase is determined before that phase
    - assignments are reconfigured periodically to restore load balance
      - e.g. Barnes-Hut where bodies move among regions
  - Dynamic tasking: Task Queue
  - Centralized task queue
    - contention
  - Distributed task queue
    - Can steal from other queues
  - In general: more overhead with dynamic methods
Task Queues

(a) Centralized task queue

(b) Distributed task queues (one per process)

Copyright 1999 Morgan Kaufmann Publishers, Inc.

Dynamic Load Balancing

Figure 3.2 Illustration of the performance impact of dynamic partitioning for load balance. The graph in (a) shows the speedups of the Barnes-Hut application with and without semistatic partitioning, and the graph in (b) shows the speedups of raytrace with and without dynamic tasking. Even in these applications that have a lot of parallelism, dynamic partitioning is important for improving load balance over static partitioning.

Copyright 1999 Morgan Kaufmann Publishers, Inc.
Impact of Task Granularity

- **Granularity** = Amount of work associated with task
- **Large tasks**
  - more difficult load balancing
  - lower overhead
  - less contention
  - less communication
- **Small tasks**
  - more load balancing possibilities
  - too much synchronization
  - too much management overhead
  - might have too much communication (use affinity scheduling)

Impact of Synchronization and Serialization

- **Too coarse synchronization**
  - barriers instead of point-to-point synch
  - poor load balancing
- **Too many synchronization operations**
  - e.g. lock each element of array
  - more difficult to program
  - too many synchronization operations
- **Coarse grain locking**
  - lock entire array
  - serialize access to array
Example: Task Queue

- Add a task to queue, search queue for another task, remove task from queue
- Option 1:
  - Make whole process one critical section
- Option 2:
  - critical section: add task to queue
  - non-critical section: search queue for another task
  - critical section: remove task from queue
    » (if task is still there)
- General guideline
  - searching (reading) does not have to be in critical section
  - updating does have to be in critical section
- Programming “trick”
  - first check in non-critical section
  - then lock and re-verify

Architectural Support for Dynamic Task Stealing

- How can architecture help?
- Communication
  - support for transfer of small amount of data and mutual exclusion
  - can make tasks smaller
  - better load balance
- Naming
  - make it easy to name or access data associated with stolen task
- Synchronization
  - support point-to-point synchronization
  - better load balancing
Reducing Inherent Communication

• Communication required for parallel program
• Communication to Computation Ratio
  – bytes / time or bytes / instruction
• Affected by assignment (task -> process)
  – assign heavily communicating tasks to same process
• Domain decomposition
  – interact with neighbors in space
  – good for simulation of physical systems

![Communicated Values](Diagram of Communicated Values)

Domain Decomposition, contd.

• Communication grows with surface
• Computation grows with volume
• Shape of partitions is application and architecture dependent
  – “squares”, row blocks, interleaved rows, etc.

![Communicated Values](Diagram of Communicated Values)
**Speedup Revisited**

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work on any processor})}
\]

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work + Synch Wait + Communication})}
\]

---

**Reducing Extra Work**

- **Redundant Computation**
  - if node would be idle anyway, compute data to avoid communication
  - e.g. at startup all processes compute shared table
    - vs one computes and then communicates to others
- **Creating processes (high cost)**
  - create once and manage in application

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work + Synch Wait + Communication + extra work})}
\]
**Multiprocessor as an Extended Mem. Hier.**

- **Example:**
  - computation: 2 instructions per cycle (IPC)
    - or 500 cycles per 1000 instructions
  - 1 long miss per 1000 instructions
    - 200 cycles per long miss
  - => 700 cycles per 1000 instructions (40% slowdown)

**Inherent vs. Artifactual Communication**

- Poor allocation of data
  - many accesses to remote nodes
- Unnecessary data in transfer
- Unnecessary data transfer because of system granularity
  - e.g. cache line sizes larger than inherent data transfer
- Redundant communication
- Limited capacity for replication
- Communication Structure
  - large vs. small messages
  - bursty
  - overlap
  - do communication patterns match the network structure (mapping)
Cache Memory 101

- **STOP HERE:**
- **Locality + smaller HW is faster = memory hierarchy**
  - *Levels:* each smaller, faster, more expensive/byte than level below
  - *Inclusive:* data found in top also found in the bottom
- **Definitions**
  - *Upper* is closer to processor
  - *Block:* minimum unit of data present or not in upper level
  - *Frame:* HW (physical) place to put block (same size as block)
  - *Address = Block address + block offset address*
  - *Hit time:* time to access upper level, including hit determination
- **3C Model**
  - compulsory, capacity, conflict
- **Add another C: communication misses**

### Cache Coherent Shared Memory

![Cache Coherent Shared Memory Diagram](image)

(C) 2001 Mark D. Hill from Adve, Falsafi, Lebek, Reinhardt & Singh CS/ECE 757
Cache Coherent Shared Memory

Orchestration for Performance

- Exploit Temporal and Spatial Locality
  - Temporal locality affects replication
  - Touch too much data == capacity misses
- Computation Blocking
**Spatial Locality**

- Communication grain
- Allocation grain
- Coherence grain (for CC shared memory)
- What benefit do you get from larger block size
- Potential disadvantage is false sharing
- Two or more processors accessing same cache block but don’t share any of the data.

---

**Poor Data Allocation**

Elements on Same Page

Elements on Same Cache Block
Data Blocking

Elements on Same Page

Elements on Same Cache Block

Data Structuring and Performance

(a) Ocean with 514 x 514 grids
(b) Equation solver kernel with 12K x 12K grid
Reducing Communication Cost

Cost = Freq. x (Overhead + Latency + Xfer size/BW - Overlap)

- Reduce Overhead
  - Fewer, larger messages
    » easier with message passing (more control over messages)
    » easier with regular data access (e.g. send entire row in solver)
- Reduce Delay (latency)
  - Depends on hardware
    » use pipelined networks (not store and forward)
    » reduce number of hops (generally not considered important today)
- Reduce Contention (bandwidth)
  - Resources have nonzero occupancy
  - Difficult to program for
  - Can cause bottlenecks (and under utilization elsewhere)
  - Endpoint contention (e.g. memory banks)
  - Network contention (e.g. interconnect net)

Reducing Communication Cost, Contd.

- Hotspots
  - Consider global sum
    » one processor vs tree of processors
- Reducing Ovelap
  - Pre-Communicating (like prefetching)
  - Non-blocking communication (do something else and come back)
  - Multi-threading (assign multiple tasks to same process and switch)
Review: Programming for Performance

• Partitioning for Performance
  – Identify concurrency
  – Managing concurrency
    » static
    » dynamic
  – Granularity of concurrency
  – Serialization and synchronization costs
  – Communication

• Orchestration for Performance
  – Exploit Locality
  – Data and Computation Blocking
  – Match system (page size, cache block size)