Steps in Creating a Parallel Program

- 4 steps: Decomposition, Assignment, Orchestration, Mapping
- Performance Goal of the steps: Maximize parallel speedup (minimize resulting parallel) execution time by:
  1. Balancing computations and overheads on processors (every processor does the same amount of work + overheads).
  2. Minimizing communication cost and other overheads associated with each step.

(Parallel Computer Architecture, Chapter 3)
Parallel Programming for Performance

A process of Successive Refinement of the steps

• **Partitioning for Performance:**
  – Load Balancing and Synchronization Wait Time Reduction
  – Identifying & Managing Concurrency
    • Static Vs. Dynamic Assignment
    • Determining Optimal Task Granularity
    • Reducing Serialization / Synch Wait Time
  – Reducing Inherent Communication
    • Minimizing *communication to computation ratio* C-to-C Ratio
  – Efficient Domain Decomposition
    Or Domain Partitioning
  – Reducing Additional Overheads

• **Orchestration/Mapping for Performance:**
  – Extended Memory-Hierarchy View of Multiprocessors For SAS
    • Exploiting Spatial Locality/Reduce Artifactual Communication
    • Structuring Communication
    • Reducing Contention
    • Overlapping Communication

(Parallel Computer Architecture, Chapter 3)
Successive Refinement of Parallel Program Performance

Partitioning is possibly independent of architecture, and may be done first (initial partition):

- View machine as a collection of communicating processors
  - Balancing the workload across tasks/processes/processors.
  - Reducing the amount of inherent communication.
  - Reducing extra work to find a good assignment.

- Above three issues are conflicting.

Then deal with interactions with architecture (Orchestration, Mapping):

- View machine as an extended memory hierarchy:
  - Reduce artifactual (extra) communication due to architectural interactions.
  - Cost of communication depends on how it is structured (possible overlap with computation) + Hardware Architecture

- This may inspire changes in partitioning.

And algorithm?
Partitioning for Performance

1. **Balancing the workload across tasks/processes:**
   - Reducing wait time at synchronization points needed to satisfy data dependencies among tasks.

2. **Reduce Overheads:**
   - Reducing interprocess inherent communication.
   - Reducing extra work needed to find a good assignment.

The above goals lead to two extreme trade-offs:

- Minimize communication => run on 1 processor.
  
  => extreme load imbalance.

  And other parallelization overheads

- Maximize load balance => random assignment of tiny tasks.
  
  => no control over communication.

- A good partition may imply extra work to compute or manage it

• The goal is to *compromise* between the above extremes
Load Balancing and “Synch Wait Time” Reduction

Limit on speedup:

\[
\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\text{Max (Work on any Processor)}}
\]

- Work includes computing, data access and other costs.
- Not just equal work, but must be busy (computing) at same time to minimize synchronization wait time to satisfy dependencies.

Four parts to load balancing and reducing synch wait time:

1. Identify enough concurrency in decomposition.
2. Decide how to manage the concurrency (statically or dynamically).
3. Determine the granularity (task grain size) at which to exploit it.
4. Reduce serialization and cost of synchronization.
Identifying Concurrency: Decomposition

• Concurrency may be found by:
  1. Examining loop structure of sequential algorithm.
  2. Fundamental data dependencies (dependency analysis/graph).
  3. Exploit the understanding of the problem to devise parallel algorithms with more concurrency (e.g. ocean equation solver).

• Software/Algorithm Parallelism Types:
  1 - Data Parallelism versus 2 - Functional Parallelism:

1 - Data Parallelism:
  – Similar parallel operation sequences performed on elements of large data structures
    • (e.g. ocean equation solver, pixel-level image processing)
  – Such as resulting from parallelization of loops.
  – Usually easy to load balance. (e.g. ocean equation solver)
  – Degree of concurrency usually increase with input or problem size. e.g. $O(n^2)$ in equation solver example.
Identifying Concurrency (continued)

2- Functional Parallelism:

• Entire large tasks (procedures) with possibly different functionality that can be done in parallel on the same or different data.
  
  e.g. different independent grid computations in Ocean.
  
  – **Software Pipelining:** Different functions or software stages of the pipeline performed on different data:
    
    • As in video encoding/decoding, or polygon rendering.
    
    • **Concurrency degree usually modest** and does not grow with input size
    
    – **Difficult to load balance.**
    
    – Often used to reduce synch wait time between data parallel phases.

Most scalable parallel programs:

(more concurrency as problem size increases) parallel programs:

  **Data parallel programs** (per this loose definition)

  – Functional parallelism can still be exploited to reduce synchronization wait time between data parallel phases.

Software/Algorithm Parallelism Types: were also covered in lecture 3 slide 33
Managing Concurrency: Task Assignment

Goal: Obtain an assignment with a good load balance among tasks (and processors in mapping step)

Static versus Dynamic Assignment:

Static Assignment: (e.g. equation solver)
- Algorithmic assignment usually based on input data; does not change at run time.
- Low run time overhead.
- Computation must be predictable.
- Preferable when applicable (lower overheads).

Dynamic Assignment: Or tasking
- Needed when computation not fully predictable.
- Adapt partitioning at run time to balance load on processors.
- Can increase communication cost and reduce data locality.
- Can increase run time task management overheads.

Example 2D Ocean Equation Solver

Counts as extra work
Dynamic Assignment/Mapping

Profile-based (semi-static):
- Profile (algorithm) work distribution initially at runtime, and repartition dynamically.
- Applicable in many computations, e.g. Barnes-Hut, (simulating galaxy evolution) some graphics.

Dynamic Tasking:
- Deal with unpredictability in program or environment (e.g. Ray tracing)
  - Computation, communication, and memory system interactions
  - Multiprogramming and heterogeneity of processors
  - Used by runtime systems and OS too.
- Pool (queue) of tasks: Processors take and add tasks to pool until parallel computation is done.
- e.g. “self-scheduling” of loop iterations (shared loop counter).
Simulating Galaxy Evolution
(Gravitational N-Body Problem)

- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
  - $O(n^2)$ brute force approach
- Hierarchical Methods (e.g. Barnes-Hut) take advantage of force law: $G \ (\text{center of mass}) \ \frac{m_1m_2}{r^2}$

- Many time-steps, plenty of concurrency across stars within one
Gravitational N-Body Problem: Barnes-Hut Algorithm

- To parallelize problem: Groups of bodies partitioned among processors. Forces communicated by messages between processors.
  - Large number of messages, \( O(N^2) \) for one iteration.
- Solution: Approximate a cluster of distant bodies as one body with their total mass
- This clustering process can be applies recursively.
- Barnes_Hut: Uses divide-and-conquer clustering. For 3 dimensions:
  - Initially, one cube contains all bodies
  - Divide into 8 sub-cubes. (4 parts in two dimensional case).
  - If a sub-cube has no bodies, delete it from further consideration.
  - If a cube contains more than one body, recursively divide until each cube has one body
  - This creates an oct-tree which is very unbalanced in general.
  - After the tree has been constructed, the total mass and center of gravity is stored in each cube.
  - The force on each body is found by traversing the tree starting at the root stopping at a node when clustering can be used.
  - The criterion when to invoke clustering in a cube of size \( d \times d \times d \):
    \[ r \geq \frac{d}{\theta} \]
    - \( r \) = distance to the center of mass
    - \( \theta \) = a constant, 1.0 or less, opening angle
  - Once the new positions and velocities of all bodies is computed, the process is repeated for each time period requiring the oct-tree to be reconstructed (repartition dynamically)
Two-Dimensional Barnes-Hut

Recursive Division of Two-dimensional Space

Locality Goal:
Bodies close together in space should be on same processor
Barnes-Hut Algorithm

- Main data structures: array of bodies, of cells, and of pointers to them
  - Each body/cell has several fields: mass, position, pointers to others
  - Pointers are assigned to processes

```
<table>
<thead>
<tr>
<th>Time-steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute forces</td>
</tr>
<tr>
<td>Update properties</td>
</tr>
</tbody>
</table>
```

```
| Build tree |
| Compute moments of cells |
| Traverse tree to compute forces |
```
The Need For Dynamic Tasking:
Rendering Scenes by Ray Tracing

- Shoot rays into a scene through pixels in image plane.
- Follow their paths: To/from light sources/reflected light
  - They bounce around as they strike objects:
    - They generate new rays:
      • Resulting in a ray tree per input ray and thus more computations (tasks).
- Result is color and opacity for that pixel.
- Parallelism across rays.
  • Parallelism here is unpredictable statically.
  • Dynamic tasking needed for load balancing.
Dynamic Tasking with Task Queues

Centralized versus distributed queues.

Task stealing with distributed queues.

- Can compromise communication and data locality (e.g. in SAS), and increase synchronization wait time.
- Whom to steal from, how many tasks to steal, ...
- Termination detection (all queues empty).
- Load imbalance possible related to size of task.
  - Many small tasks usually lead to better load balance
Performance Impact of Dynamic Assignment

On SGI Origin 2000 (cache-coherent shared distributed memory):

Barnes-Hut 512k particle
(N-Body Problem)

Ray tracing

NUMA
Assignment: Determining Task Granularity

Recall that parallel task granularity:

Amount of work or computation associated with a task.

General rule:

- **Coarse-grained** => Often less load balance
  - Larger/fewer tasks
  - Less communication and other overheads

- **Fine-grained** => More overhead; often more
  - Smaller/more tasks
  - Communication, contention

Communication, contention actually more affected by

**mapping to processors, not just task size only.**

- Other overheads are also affected by task size too, particularly
  with dynamic mapping (tasking) using task queues:
  - Small tasks => More Tasks => More dynamic mapping overheads.

A task only executes on one processor to which it has been mapped or allocated
Reducing Serialization/Synch Wait Time

Requires careful assignment and orchestration (and scheduling ?)

Reducing Serialization/Synch wait time in Event synchronization:

- Reduce use of conservative synchronization e.g.:
  - Fine point-to-point synchronization instead of barriers (if possible),
  - or reduce granularity of point-to-point synchronization (specific elements instead of entire data structure).

- But fine-grained synch more difficult to program, more synch operations.

Reducing Serialization in Mutual exclusion:

1. Separate locks for separate data
   - e.g. locking records in a database instead of locking entire database: lock per process, record, or field
   - Lock per task in task queue, not per queue
   - Finer grain => less contention/serialization, more space, less reuse

2. Smaller, less frequent critical sections
   - No reading/testing in critical section, only modification
   - e.g. searching for task to dequeue in task queue, building tree etc.

3. Stagger critical sections in time (on different processors).
   - i.e critical section entry occur at different times

Partitioning for Performance:
Implications of Load Balancing/Synch Time Reduction

Extends speedup limit expression to:

\[ Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{\max (\text{Work} + \text{Synch Wait Time})} \]

Generally load balancing is the responsibility of software

Architecture can support task stealing and synch efficiently:

- **Fine-grained** communication, low-overhead access to queues
  - Efficient support allows smaller tasks, better load balancing
- **Naming** logically shared data in the presence of task stealing
  - Need to access data of stolen tasks, esp. multiple-stolen tasks

\[ \Rightarrow \text{Hardware shared address space advantageous here} \]

- Efficient support for point-to-point communication.
  - Software layers + hardware (network) support.
Reducing Inherent Communication

**Measure:** communication to computation ratio (c-to-c ratio)

Focus here is on reducing interprocess communication inherent in the problem:

- Determined by assignment of parallel computations to tasks/processes.
- Minimize c-to-c ratio while maintaining a good load balance among tasks/processes.
- Actual communication can be greater than inherent communication.

- As much as possible, assign tasks that access same data to same process (and processor later in mapping).
- Optimal solution (partition) to reduce communication and achieve an optimal load balance is NP-hard in the general case.
- Simple heuristic partitioning solutions may work well in practice:
  - Due to specific dependency structure of applications.
  - Example: Domain decomposition or domain partitioning
Example Assignment/Partitioning Heuristic:

**Domain Decomposition**

- Initially used in **data parallel** scientific computations such as (Ocean) and pixel-based image processing to obtain a good load balance and c-to-c ratio. 
  
- The task assignment is achieved by **decomposing the physical domain or data set of the problem.**

- Exploits the local-biased nature of physical problems
  - Information requirements often short-range
  - Or long-range but fall off with distance

- Simple example: Nearest-neighbor 2D grid computation (as in ocean example)

  ![Diagram of Block Decomposition](image)

  Communication = \( \frac{4n}{\sqrt{p}} \)
  
  Computation = \( \frac{n^2}{p} \)

  \( C_{\text{to-com}} = \frac{4 \times \sqrt{p}}{n} \)

  The **comm-to-comp ratio** = Perimeter to Area (area to volume in 3-d)

  - Depends on \( n, p \): decreases with \( n \), increases with \( p \)

\[ p = \text{Number of tasks/processes} \quad \text{here} = p = 4 \times 4 = 16 \]
Domain Decomposition (continued)

Best domain decomposition depends on information requirements

Nearest neighbor example: i.e group or “strip” of (contiguous) rows

- block versus strip domain decomposition:

For block assignment

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>P5</td>
<td>P6</td>
<td>P7</td>
</tr>
<tr>
<td>P8</td>
<td>P9</td>
<td>P10</td>
<td>P11</td>
</tr>
<tr>
<td>P12</td>
<td>P13</td>
<td>P14</td>
<td>P15</td>
</tr>
</tbody>
</table>

Comm. \( \frac{4n}{\sqrt{p}} \)
Comp. \( \frac{n^2}{p} \)

For strip assignment

<table>
<thead>
<tr>
<th>Strip assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/p rows per task</td>
</tr>
</tbody>
</table>

Communication = 2n
Computation = \( \frac{n^2}{p} \)
c-to-c ratio = \( \frac{2p}{n} \)

Which C-to-C ratio is better?

Application dependent: strip may be better in some cases

Often \( n \gg p \)
Finding a Domain Decomposition

Four possible methods:

1. **Static, by inspection:**
   - Computation must be predictable: e.g. grid example above, and Ocean
   - Not input data dependent

2. **Static, but not by inspection:**
   - Input-dependent, require analyzing input structure
     - Before start of computation once input data is known.
     - E.g. sparse matrix computations, data mining

3. **Semi-static (periodic repartitioning):**
   - Characteristics change but slowly; e.g. Barnes-Hut

4. **Static or semi-static, with dynamic task stealing**
   - Initial decomposition based on domain, but highly unpredictable computation; e.g. ray tracing

Characterized by non-uniform data/computation distribution
Implications of Communication

- Architects must examine application latency/bandwidth needs
- If denominator in c-to-c is computation execution time, ratio gives average BW needs per task.
- If denominator in c-to-c is operation count, gives extremes in impact of latency and bandwidth
  - Bandwidth: assume all latency hidden.
  - Reality is somewhere in between.
- Actual impact of communication depends on structure and cost as well:
  \[
  \text{Speedup} \leq \frac{\text{Sequential Work}}{\text{Max} \left( \text{Work + Synch Wait Time + Comm Cost} \right)}
  \]
  
  → Need to keep communication balanced across processors as well.

\( c-to-c = \text{communication to computation ratio} \)

\( \text{Communication Cost} = \text{Time added to parallel execution time as a result of communication} \)
Partitioning for Performance: Reducing Extra Work (Overheads)

• **Common sources of extra work (mainly orchestration):**
  – Computing a good partition (at run time):
    e.g. partitioning in Barnes-Hut or sparse matrix
  – Using **redundant** computation to avoid communication.
  – Task, data distribution and process management overhead
    • Applications, languages, runtime systems, OS
  – Imposing **structure** on communication:
    • Coalescing (combining) messages, allowing effective naming

• **Architectural Implications:**
  – Reduce by making communication and orchestration efficient
    (e.g. hardware support of primitives ?)

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\text{Max (Work + Synch Wait Time + Comm Cost + Extra Work)}}
\]

(on any processor)
Summary of Parallel Algorithms Analysis

- Requires characterization of multiprocessor system and algorithm requirements.
- **Historical focus on algorithmic aspects:** partitioning, mapping
- In **PRAM model:** data access and communication are *free*
  - Only load balance (including serialization) and extra work matter
  + Synch Wait Time

For **PRAM:**

\[
\text{Speedup} \leq \frac{\text{Sequential Instructions}}{\text{Max} (\text{Instructions} + \text{Synch Wait Time} + \text{Extra Instructions})}
\]

- Useful for parallel algorithm development, but possibly unrealistic for real parallel program performance evaluation.
  - Ignores communication and also the imbalances it causes
  - Can lead to poor choice of partitions as well as orchestration when targeting real parallel systems.
Limitations of Parallel Algorithm Analysis

Artifactual “extra” communication caused by program implementation and architectural interactions can even dominate.

Thus, actual amount of communication may not be dealt with adequately.

Cost of communication determined not only by amount:

- Also how communication is structured and overlapped.
- Cost of communication (primitives) in system
  
  - Software related and hardware related (network) including CA

Both are architecture-dependent, and addressed in orchestration step.
Generic Multiprocessor Architecture

Computing Nodes:
processor(s), memory system, plus communication assist (CA):
• Network interface and communication controller.

Scalable Network.

If SAS is natively supported by this generic architecture:
→ NUMA
(Distributed Shared memory Architecture)

Scalable network.
Nodes
CA may support SAS in hardware or just message-passing
Extended Memory-Hierarchy View of
Generic Multiprocessors

- **Levels in extended hierarchy:**
  - Registers, caches, local memory, remote memory (over network)
  - **Glued together by communication architecture**
  - Levels communicate at a certain granularity of data transfer. (e.g. Cache blocks, pages etc.)

- **Need to exploit spatial and temporal locality in hierarchy**
  - Otherwise artifactual (extra) communication may also be caused

Why?

- Especially important since communication is expensive

This extended hierarchy view is more useful in distributed shared memory (NUMA) parallel architectures

---

CMPE655 - Shaaban

For NUMA SAS Parallel Architectures

SAS support Assumed

1 Registers, 2 caches, 3 local memory, 4 remote memory (over network)

i.e Minimum size of data transferred between levels

Over network

#29 lec # 6 Spring 2014 3-18-2014
Extended Hierarchy

- **Idealized view**: local cache hierarchy + single main memory
- **But reality is more complex**:
  - **Centralized Memory**: + caches of other processors
  - **Distributed Memory**: some local, some remote; + network topology + local and remote caches
- **Management of levels**:
  - Caches managed by hardware
  - Main memory depends on programming model:
    - SAS: data movement between local and remote transparent
    - Message passing: explicit by sending/receiving messages.
- Improve performance through **architecture or program locality** (maximize local data access).

Otherwise artifactual “extra” communication is created

This extended hierarchy view is more useful in distributed shared memory parallel architectures
Artifactual Communication in Extended Hierarchy

Accesses not satisfied in local portion cause communication

- **Inherent Communication**, implicit or explicit, causes transfers:
  - Determined by parallel algorithm/program partitioning

- **Artifactual “Extra” Communication**:
  - Determined by program implementation and architecture interactions
  - Poor allocation of data across distributed memories: data accessed heavily used by one node is located in another node’s local memory.
  - **Unnecessary data in a transfer**: More data communicated in a message than needed.
  - **Unnecessary transfers due to system granularities** (cache block size, page size).
  - **Redundant communication of data**: data value may change often but only last value needed.
  - **Finite replication capacity** (in cache or main memory)
    - Inherent communication assumes unlimited capacity, small transfers, perfect knowledge of what is needed.
    - More on artifactual communication later; first consider replication-induced further

As defined earlier: **Inherent Communication** : communication between tasks inherent in the problem/parallel algorithm for a given partitioning/assignment (to tasks)
Extra Communication and Replication

• Extra Comm. induced by finite capacity is most fundamental artifact:
  – Similar to cache size and miss rate or memory traffic in uniprocessors.
  – Extended memory hierarchy view useful for this relationship
• View as three level hierarchy for simplicity
  – Local cache, local memory, remote memory (ignore network topology).
• Classify “misses” in “cache” at any level as for uniprocessors
  1. Compulsory or cold misses (no size effect)
  2. Capacity misses (yes)
  3. Conflict or collision misses (yes)
  4. Communication or coherence misses (no)
  – Each may be helped/hurt by large transfer granularity (spatial locality).

Extra Communication and Replication

replication i.e. cache is filled to capacity

4 Cs

i.e misses that result in extra communication over the network

Distributed shared memory (NUMA) parallel architecture implied here
Working Set Perspective

The data traffic between a cache and the rest of the system and components data traffic as a function of cache size

- Hierarchy of working sets
- Traffic from any type of miss can be local or non-local (communication)

Distributed shared memory/SAS parallel architecture assumed here
Orchestration for Performance

• Reducing amount of communication:
  – **Inherent**: change logical data sharing patterns in algorithm
    • Reduce c-to-c-ratio.
  – **Artifactual**: exploit spatial, temporal locality in extended hierarchy.
    • Techniques often similar to those on uniprocessors

• **Structuring communication to reduce cost:**
  e.g. overlap communication with computation or other communication
• We’ll examine techniques for both...
Reducing Artifactual Communication

- **Message Passing Model:**
  - Communication and replication are both explicit.
  - Even artifactual communication is in explicit messages
    - e.g. more data sent in a message than actually needed

- **Shared Address Space (SAS) Model:**
  - More interesting from an architectural perspective
  - Occurs transparently due to interactions of program and system:
    - Caused by sizes of allocation and granularities in extended memory hierarchy (e.g. Cache block size, page size).

- Next, we use shared address space to illustrate issues
  - (distributed memory SAS)

Orchestration for Performance:

i.e. Artifactual Comm.
Exploiting Temporal Locality

- Structure algorithm so working sets map well to hierarchy
  - Often techniques to reduce inherent communication do well here
  - Schedule tasks for data reuse once assigned
- Multiple data structures in same phase
  - e.g. database records: local versus remote
- Solver example: blocking (or blocked data access pattern)

Better Temporal Locality

- More useful when $O(n^{k+1})$ computation on $O(n^k)$ data
  - Many linear algebra computations (factorization, matrix multiply)

Reducing Artifactual “Extra” Communication

i.e. 2D Grid Computation

Unblocked Data Access Pattern

(a) Unblocked access pattern in a sweep

(b) Blocked access pattern with $B = 4$

Blocked Data Access Pattern

i.e computation with data reuse

Blocked assignment assumed here
Exploiting Spatial Locality

- Besides capacity, granularities are important:
  - Granularity of allocation (e.g. page size)
  - Granularity of communication or data transfer
  - Granularity of coherence (e.g. cache block size)

- Major spatial-related causes of artifactual communication:
  - Conflict misses
  - Data distribution/layout (allocation granularity)
  - Fragmentation (communication granularity)
  - False sharing of data (coherence granularity)

- All depend on how spatial access patterns interact with data structures/architecture:
  - Fix problems by modifying data structures, or layout/alignment (as shown in example next)

- Examine later in context of architectures
  - One simple example here: data distribution in SAS solver

Reducing Artifactual “Extra” Communication

Larger “granularity” when farther from processor

- Fix?

Distributed memory (NUMA) SAS assumed here

CMPE655 - Shaaban
Spatial Locality Example

- Repeated sweeps over elements of 2D grid, block assignment, Shared address space;
- In Distributed memory: A memory page is allocated in one node’s memory;
- Natural 2D versus higher-dimensional (4D here) array representation.

**Ex:** (1024, 1024)  
**Ex:** (4, 4, 256, 256)

Contiguity in memory layout

Two-Dimensional (2D) Array

- Page straddles partition boundaries: difficult to distribute memory well.
- Cache block straddles partition boundary.

Four-Dimensional (4D) Array

- Page does not straddle partition boundary.
- Cache block is within a partition.

Reducing Artifactual “Extra” Communication

- (SAS) i.e granularity of data allocation

(SAS) assumed here  
Performance Comparison Next
Execution Time Breakdown for Ocean on a 32-processor Origin2000

1026 x 1026 grids with block partitioning on 32-processor Origin2000

- 4D grids much better than 2D, despite very large caches on machine (4MB L2 cache)
  - data distribution is much more crucial on machines with smaller caches
- Major bottleneck in this configuration is time waiting at barriers
  - imbalance in memory stall times as well

**Speedup** = $6/3.5 = 1.7$

Four-dimensional (4D) arrays

Two-dimensional (2D) arrays

Thus less replication capacity
Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows (i.e. strip assignment)

- Blocks still have a spatial locality problem on remote data
- Row-wise (strip) can perform better despite worse inherent c-to-c ratio

• Result depends on \( n \) and \( p \)

Comm-to-comp ratio: \( \frac{4\times\sqrt{p}}{n} \) for block, \( \frac{2\times p}{n} \) for strip

Results to show this next
Example Performance Impact

Equation solver on SGI Origin2000 (distributed shared memory)

- \( rr = \) Round Robin Page Distribution
- Rows = Strip Assignment

![Graph showing speedup versus number of processors for different block and strip assignments.](#)

### 514 x 514 grids

- Ideal Speedup
- Comm-to-comp ratio: \( \frac{4 \sqrt{p}}{n} \) for block, \( \frac{2 \times p}{n} \) for strip

### 12k x 12k grids

- 2D, 4D block assignment
- 2D, 4D-rr
- Rows
- Rows-rr

**Why?**

- Super-linear Speedup

---

#41 lec #6 Spring 2014 3-18-2014
Structuring Communication

Given amount of comm. (inherent or artifactual), goal is to reduce cost ⬇️

- Total cost of communication as seen by process:

\[ C = f \times (o + l + \frac{n_c/m}{B} + t_c - \text{overlap}) \]

- \( f \) = frequency of messages
- \( o \) = overhead per message (at both ends)
- \( l \) = network delay per message
- \( n_c \) = total data sent
- \( m \) = number of messages
- \( B \) = bandwidth along path (determined by network, NI, assist)
- \( t_c \) = cost induced by contention per message
- \( \text{overlap} \) = amount of latency hidden by overlap with comp. or other comm.

- Portion in parentheses is **cost of a message** (as seen by processor)
- That portion, ignoring overlap, is **latency of a message**
- **Goal:** 1- reduce terms in communication latency and 2- increase overlap

Communication Cost: Actual time added to parallel execution time as a result of communication
Reducing Overall Communication Overhead

Can reduce number of messages $f$ or reduce overhead per message $o$.

Message overhead, $o$ is usually determined by hardware and system software (implementation cost of comm. primitives).

- Program should try to reduce number of messages $m$ by combining messages.
- More control when communication is explicit (message-passing).

Combining data into larger messages:

- Easy for regular, coarse-grained communication
- Can be difficult for irregular, naturally fine-grained communication.

  - May require changes to algorithm and extra work
    - Combining data and determining what and to whom to send
  - May increase synchronization wait time.

    Longer synch wait to get more results data computed to send in larger message

Reduce total comm. overhead, How?

Reducing Cost of Communication:

$\textit{f} \ast \textit{o}$

CMPE655 - Shaaban
Reducing Network Delay

Total Number of Messages

- Total network delay component = \( f \times l = f \times h \times t_h \)
  - \( h \) = number of hops traversed in network
  - \( t_h \) = link+switch latency per hop

- Reducing \( f \): Communicate less, or make messages larger
- Reducing \( h \) (number of hops):
  - Map task communication patterns to network topology
    e.g. nearest-neighbor on mesh and ring etc.
  - How important is this?
    - Used to be a major focus of parallel algorithm design
    - Depends on number of processors, how \( t_h \), compares with other components, network topology and properties
    - Less important on modern machines
      - (e.g. Generic Parallel Machine)

Where equal communication time/delay between any two nodes is assumed (i.e. symmetric network)

Reducing Cost of Communication:

Thus fewer messages
Mapping of Task Communication Patterns to Topology

Example (network)

Parallel System Topology: 3D Binary Hypercube

Task Graph:

- T1 runs on P0
- T2 runs on P5
- T3 runs on P6
- T4 runs on P7
- T5 runs on P0

Poor Mapping:

- Communication from T1 to T2 requires 2 hops
  Route: P0-P1-P5
- Communication from T1 to T3 requires 2 hops
  Route: P0-P2-P6
- Communication from T1 to T4 requires 3 hops
  Route: P0-P1-P3-P7
- Communication from T2, T3, T4 to T5
  • similar routes to above reversed (2-3 hops)

Better Mapping:

- T1 runs on P0
- T2 runs on P1
- T3 runs on P2
- T4 runs on P4
- T5 runs on P0

- Communication between any two communicating (dependant) tasks requires just 1 hop

Reducing Network Delay: Reduce Number of Hops
Reducing Contention

- All resources have nonzero occupancy (busy time):
  - Memory, communication assist (CA), network link, etc.
  - Can only handle so many transactions per unit time.
  - Contention results in queuing delays at the busy resource.

- Effects of contention:
  - Increased end-to-end cost for messages. (e.g., delay, latency)
  - Reduced available bandwidth for individual messages.
  - Causes imbalances across processors.

- Particularly insidious performance problem:
  - Easy to ignore when programming
  - Slows down messages that don’t even need that resource
    - By causing other dependent resources to also congest
  - Effect can be devastating: Don’t flood a resource!

Reducing Cost of Communication:

- \( t_c \)
- i.e., Occupancy
- i.e., contended

Ripple effect

How?
Types of Contention

• Network contention and end-point contention (*hot-spots*)

• **Location and Module** Hot-spots:
  – **Location**: e.g. accumulating into global variable, barrier
    • **Possible solution**: tree-structured communication
  – **Module**: all-to-all personalized comm. in matrix transpose
    • **Solution**: stagger access by different processors to same node temporally

• In general, reduce burstiness (smaller messages); may conflict with making messages larger (to reduce number of messages)
Overlapping Communication

• Cannot afford to stall/wait for high latencies
• Overlap with computation or other communication to hide latency → To reduce communication cost

• Common Techniques:
  1 – Prefetching (start access or communication before needed)
  2 – Block data transfer (may introduce extra communication)
  3 – Proceeding past communication (e.g. non-blocking receive)
  4 – Multithreading (switch to another ready thread or task)

• In general these above techniques require:
  1 – Extra concurrency per node (slackness) to find some other computation.
  2 – Higher available network bandwidth (for prefetching).
  3 – Availability of communication primitives that support overlap.

More on these techniques in PCA Chapter 11
**Summary of Tradeoffs**

- **Different goals often have conflicting demands:**
  - **Better Load Balance Implies:**
    - Fine-grain tasks
    - Random or dynamic assignment
  - **Lower Amount of Communication Implies:**
    - Usually coarse grain tasks
    - Decompose to obtain locality: not random/dynamic
  - **Lower Extra Work Implies:**
    - Coarse grain tasks
    - Simple assignment
  - **Lower Communication Cost Implies:**
    - Big transfers: to amortize overhead and latency
    - Small transfers: to reduce contention

So, big or small date transfers ??
Relationship Between Perspectives

<table>
<thead>
<tr>
<th>Parallelization step(s)</th>
<th>Performance issue</th>
<th>Processor time component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition/assignment/orchestration</td>
<td>Load imbalance and synchronization</td>
<td>Synch wait</td>
</tr>
<tr>
<td>Decomposition/assignment</td>
<td>Extra work</td>
<td>Busy-overhead</td>
</tr>
<tr>
<td>Decomposition/assignment</td>
<td>Inherent communication volume</td>
<td>Data-remote</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Artifactual communication and data locality</td>
<td>Data-local</td>
</tr>
<tr>
<td>Orchestration/mapping</td>
<td>Communication structure</td>
<td></td>
</tr>
</tbody>
</table>
Components of Execution Time From Processor Perspective

(a) Sequential

(b) Parallel with four proc

(i.e synch wait time)

(i.e comm. time)

Synchronization
Busy-overhead
Data-remote
Data-local

Data-remote
Busy-useful

(Here perfect load balancing shown)
Summary

$$Speedup_{prob}(p) = \frac{Busy(1) + Data(1)}{\max( Busy_{useful}(p) + Data_{local}(p) + Synch(p) + Date_{remote}(p) + Busy_{overhead}(p) )}$$

- Goal is to reduce denominator components
- Both programmer and system have a role to play
- Architecture cannot do much about load imbalance or too much communication
- But it can help:
  - Reduce incentive for creating ill-behaved programs (efficient naming, communication and synchronization)
  - Reduce artifactual communication
  - Provide efficient naming for flexible assignment
  - Allow effective overlapping of communication