Limitations to Parallel Speedup

As we have seen, there are many obstacles on the road to speedup:

\[
\text{Speedup}_{\text{broken}}(p) \leq \frac{\text{Sequential Work}}{\text{Parallelizable Work} + \text{Non-Parallelizable Work}}
\]

Today we will take a closer look at them from a programmer’s perspective.

This all starts with the parallelization process...

Maximizing Parallel Performance

Last Time:
- Introduced some parallel performance limiters
- Covered parallelization process

Today we will look at:
- Look closer at fundamental performance limiters
- Examine effective parallelization techniques
- Discuss interactions/tradeoffs between optimization techniques

Review of Parallelization
Partitioning for Performance

Key goals in decomposition/assignment include:
- Balance the workload
- Reduce communication
- Decrease amount of extra work

We will take a closer look at these three...

Balancing The Workload

Need to balance the load AND synchronization wait time.

Limit on speedup: \( \text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\text{Max Work On Any Processor}} \)

- Work includes data access and other costs
- Not just equal work, but must be busy at same time

Keys to balancing workload:
- Identify enough concurrency.
- Decide how to manage it.
- Determine the granularity at which to exploit it.
- Reduce serialization and cost of synchronization.

Identifying Concurrency

Two key concepts:
- **data parallelism** – identical operations performed on different items of same dataset
  
  ```
  for(i = 0; i < n; i++)
  a[i] = b[i] + c[i];
  ```

- **function parallelism** – different operations carried out on possibly diverse datasets
  
  ```
  for(i = 1; i < n; i++)
  a[i] = a[i-1] + b[i];
  ```

  ```
  ... for(i=1; i<n; i++)
  c[i] = c[i-1] + b[n-i] * k;
  ```

Partition to exploit either of these types of parallelism.

Managing Concurrency

Assignment can be done either statically or dynamically.

Static:
- Algorithmic assignment based on input; won’t change.
- Low runtime overhead.
- Computation must be predictable.

Dynamic:
- Adapt at runtime to balance load.
- Can increase communication and reduce locality.
- Can increase task management overheads.
Static Assignment

Parallelizing agent decides how to divide computational tasks among processes (threads).

Easy to do if the computations are regular and predictable.

Otherwise, we risk load imbalance.

Dynamic Assignment

Profile-based (semi-static):
- Profile work distribution at runtime
- Repartition dynamically

Dynamic Tasking:
- Maintain pool of tasks
- Threads take tasks until all done
- Deal with unpredictability in program or environment
- Cost of management could be significant (computation, communication, memory system interaction)

Impact of Dynamic Assignment

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

- Can compromise communication and compromise locality, and increase synchronization
- Whom to steal from, how many tasks to steal, ...
- Termination detection
- Maximum imbalance related to size of task
Determining Task Granularity

A key reoccurring concept is **granularity** — the quantity of work associated with a task.

General rule:
- Coarse-grained => often less load balance
- Fine-grained => more overhead; if not careful, more communication, and contention

Communication and contention actually affected by assignment, not size

Reducing Serialization

Try to reduce time spent waiting at synchronization points.

Closely tied to not just assignment, but also orchestration.

Choose granularity of **event synchronization**:
- Fine-grain- Reduce use of conservative synchronization (point-to-point instead of barriers).
- Course-grain- Easier to program, potentially fewer synchronization operations.

Apply better use of **mutual exclusion**:
- Separate locks for separate data.
- Smaller, less frequent critical sections.
- Stagger critical sections in time.

Implications of Load Balance

Extends speedup limit expression to: \[
\frac{\text{Sequential Work}}{\text{Max} (\text{Work} + \text{Synch Wait})}
\]

Generally, responsibility of software.

Architecture can support task stealing and synchronization efficiently
- **Fine-grained** communication, **low-overhead access** to queues
  - efficient support allows smaller tasks, better load balance
- **Naming** logically shared data in the presence of task stealing
  - need to access data of stolen tasks, esp. multiply-stolen tasks

=> Hardware shared address space advantageous
- Efficient support for point-to-point communication

Reducing Inherent Communication

Communication is so expensive, that we have to find ways of limiting it.

Important concept: **communication to computation ratio**

Focus here on inherent communication (we’ll talk about other forms later).

Largely dependent on assignment, so assign tasks that access same data to same process.
**Domain Decomposition**

Assign related tasks (within the same “domain”) to the same process.

Usually these tasks operate on the same or similar set of data.

In scientific applications, domains often map to physical dimensions.

Exploits local-biased nature of physical problems (e.g. nearest-neighbor computations).

---

**Finding a Domain Decomposition**

The right choice of domain decomposition to apply is highly specific to problem:

- **Static (Inspection)**- computations are predictable.
- **Static (Analysis)**- nature of computation is highly input-dependent.
- **Semi-static (Periodic Repartitioning)** - characteristics change slowly.
- **Static/Semi-static (Dynamic Task Stealing)** – highly unpredictable.

There are many variations on these approaches.

---

**Example: Domain Decomposition**

Best domain decomposition depends on information requirements

Nearest neighbor example: block versus strip decomposition:

```
  P0 P1 P2 P3
  P4 P5 P6 P7
  P8 P9 P10 P11
  P12 P13 P14 P15
```

- Comm to comp: $\frac{n}{\sqrt{p}}$ for block, $\frac{2np}{4\sqrt{p}}$ for strip
  - Retain block from here on

Application dependent: strip may be better in other cases
  - E.g. particle flow in tunnel

---

**Implications of Communication Cost**

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

Communication costs in a real system can be significant.
Reducing Extra Work

Along the way, we’ve already talked about ways that extra work can be introduced:

- Effort in computing a good partition.
- Task, data and process management overhead.

Sometimes, extra work can be traded-off against communication/synchronization

- Redundant local computations to avoid global communication).
- Data transformation to reduce communication/synchronization.

Summary: Analyzing Parallel Algorithms

Requires characterization of multiprocessor and algorithm

Historical focus on algorithmic aspects: partitioning, mapping

PRAM model: data access and communication are free

- Only load balance (including serialization) and extra work matter

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

Basics: Multiprocessors

Multiprocessors are a collection of communicating processors.

These are multi-cache, multi-memory systems:

- Role of these components essential regardless of programming model
- Programming model and communication abstraction affect specific performance tradeoffs.

Enough With The Algorithm Analysis!

Inherent communication in parallel algorithm is not the end of the story.

Communication cost depends on frequency and communication time:

Artifactual increase frequency; causes include:

- program implementation
- architectural interactions (can dominate)

Communication time is obviously dependent on architectural implementations.
Memory-Oriented View

Multiprocessors have an *extended memory hierarchy*:
- registers
- caches
- local memory
- remote memory

Communication architecture glues it all together.

Guiding principles are exploiting *temporal* and *spatial locality*.

Uniprocessor Performance

Obviously, performance depends heavily on memory hierarchy.

Time spent by a program:

\[ \text{Time}_{\text{prog}}(1) = \text{Busy}(1) + \text{Data Access}(1) \]

Data access time can be reduced by:
- Optimizing machine via bigger caches, reducing latency, increasing bandwidth...
- Optimizing program by exploiting temporal and spatial locality.

Multiprocessors and Extended Hierarchy

Things are necessarily more complex in the multiprocessor case.

Key Issues:
- *centralized memory*- caches of other processors
- *distributed memory*- some local, some remote, and also network topology
- *memory management*
  - caches managed by hardware
  - main memory depends on programming model
    - SAS: data movement between local and remote transparent
    - message passing: explicit

Levels closer to processor are lower latency and higher bandwidth.

Artifactual Communication in Extended Hierarchy

Accesses not satisfied in locally cause communication (both inherent and artifactual).

Artifactual communication is determined by program implementation and architecture interactions:
- poor data allocation
- unnecessary data in transfer
- unnecessary transfers due to granularity
- redundant communication
- finite replication
Communication and Replication

Communication induced by finite capacity is most fundamental artifact (like cache size/miss rate in uniprocessors).

Basic concepts apply throughout extended memory hierarchy.

Classify “misses” in “cache” at any level as for uniprocessors:
- compulsory or cold misses (no size effect)
- capacity misses (yes)
- conflict or collision misses (yes)
- communication or coherence misses (no)

Each type may be helped/hurt by large transfer granularity (spatial locality).

Working Sets

At first level cache (fully assoc, one-word block), inherent to algorithm.
Traffic from any type of miss can be local or nonlocal (communication).

Orchestration for Performance

How can better orchestration help?

First: Reduce amount of communication.
  - Inherent: change logical data sharing patterns in algorithm.
  - Artifactual: exploit spatial, temporal locality in extended hierarchy (similar to uniprocessor).

Second: When communication is necessary, structure it to reduce cost.

Reducing Artifactual Communication

As we have seen orchestration is very model dependent.

Message Passing Model
  - Communication and replication are both explicit.
  - Even artifactual communication is in explicit messages.

Shared Address Space Model
  - More interesting from an architectural perspective.
  - Occurs transparently due to interactions of program and system.

Use shared address space to illustrate issues.
Exploiting Temporal Locality

- Structure program so that same memory locations are accessed in a short time frame.
- Map working set to memory hierarchy.
- 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
- With multiple data structures, may have to choose which one to exploit temporal locality on.

Example: Temporal Locality

Blocking is an example of a temporal locality exploitation technique.

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

Exploiting Spatial Locality

Granularity plays an important role in spatial locality.

Types of granularity:
- allocation - for cache blocks and memory pages
- communication - for data transfer
- coherence - for data replication

Major spatial-related causes of artifactual communication:
- conflict misses
- data distribution/layout (allocation granularity)
- fragmentation (communication granularity)
- false sharing of data (coherence granularity)

Spatial Locality Example

- Repeated sweeps over 2-d grid, each time adding 1 to elements
- Natural 2-d versus higher-dimensional array representation

Contiguity in memory layout
Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows
- Blocks still have a spatial locality problem on remote data
- Rowwise can perform better despite worse inherent communication-to-computation ratio

Architectural Implications of Locality

Communication abstraction that makes exploiting it easy

For cache-coherent SAS, e.g.:
- Size and organization of levels of memory hierarchy
  - cost-effectiveness: caches are expensive
  - caveats: flexibility for different and time-shared workloads
- Replication in main memory useful? If so, how to manage?
  - hardware, OS/runtime, program?
- Granularities of allocation, communication, coherence (?)
  - small granularities => high overheads, but easier to program

Performance Impact

Equation solver on SGI Origin2000

Structuring Communication

Reduce cost of communication (inherent or artifactual).

Cost of communication as seen by process:

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

- \( f \) = 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 communication
Reducing Overhead

Reduce number of messages $m$ or overhead per message $o$.

$o$ is usually determined by hardware or system software.

But programmer/compiler can coalesce data into larger messages:
- Easy for regular, coarse-grained communication
- Can be difficult for irregular, naturally fine-grained communication

Reducing Network Delay

Network delay component = $f^h t_h$
- $h$ = number of hops traversed in network
- $t_h$ = link+switch latency per hop

Reducing $f$: communicate less, or make messages larger
Reducing $h$: map communications patterns to network topology

Reducing Contention

All resources have nonzero occupancy:
- Memory, communication controller, network link, etc.
- Can only handle so many transactions per unit time.

Effects of contention:
- Increased end-to-end cost for messages.
- Reduced available bandwidth for individual messages.
- Causes imbalances across processors.

Particularly insidious performance problem:
- Easy to ignore when programming.
- Slow down messages that don’t even need that resource.
- Effect can be devastating: Don’t flood a resource!

Types of Contention

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

Location and Module Hot-spots
- Location: e.g. accumulating into global variable, barrier
- 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; may conflict with making messages larger
**Overlapping Communication**

Cannot afford to stall for high latencies.

Overlap with computation or communication to hide latency.

Requires extra concurrency (*slackness*), higher bandwidth.

**Techniques:**
- Prefetching
- Block data transfer
- Proceeding past communication
- Multithreading

---

**Understanding Tradeoffs**

Different goals often have conflicting demands:

- **Load Balance**
  - fine-grain tasks
  - random or dynamic assignment

- **Communication**
  - usually coarse grain tasks
  - decompose to obtain locality: not random/dynamic

- **Extra Work**
  - coarse grain tasks
  - simple assignment

- **Communication Cost:**
  - big transfers: amortize overhead and latency
  - small transfers: reduce contention

---

**Processor-Centric Perspective**

![Graph showing time (s) for sequential and parallel execution with four processors.](image)
A Final Look At Speedup

\[
\text{Speedup}_{\text{prof}}(p) = \frac{\text{Busy}(I) + \text{Data}(I)}{\text{Busy}_{\text{useful}}(p) + \text{Data}_{\text{local}}(p) + \text{Synch}(p) + \text{Date}_{\text{remote}}(p) + \text{Busy}_{\text{overhead}}(p)}
\]

Goal is to reduce denominator components.

Both programmer and system have role to play.

Architecture cannot do much about load imbalance or too much communication.

But it can:

- 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

Summary

There are numerous types of performance limiters in parallel systems.

There are an equal number of techniques for dealing with them.

Performance issues trade off with one another.