Organization

Part I  Introduction, Terminology
Example (crash simulation)
Speedup and Efficiency, Amdahl’s Law
Architectures
Parallel Computing Hurdles
Distributed Memory Communication, MPI
Load Balancing I
Shared Memory, OpenMP
Organization cont.

Part II  Hybrid Computing
Parallelizing Serial Programs
Load Balancing II
Accelerators
Data Intensive Computing
Performance Improvement, Tools
Using Parallel Systems
Review, Wrapup
In this first half we introduce parallel computing and some useful terminology.

We examine many of the variations in system architecture, and how they affect the programming options.

We will look at a representative example of a large scientific/engineering code, and examine how it was parallelized. We also consider some additional examples.
Why use Parallel Computers?

- Parallel computers can be the only way to achieve specific computational goals
- PetaFLOPS and Petabytes for complex problems
- mega-transactions per second for search engines, ATM networks, digital multimedia

*Because you have to:* all computers are parallel, and the parallelism is increasing
Why Parallel Computing — continued

- The universe is inherently parallel, so parallel models fit it best.

- Physical processes occur in parallel: weather, galaxy formation, epidemics, traffic jams, ...

- Social/work processes occur in parallel: ant colonies, wolf packs, assembly lines, tutorials, ...

Stout and Jablonowski – p. 6/268
Basic Terminology and Concepts

Caveats

- The definitions are fuzzy, many terms are not standardized, definitions often change over time.
- Many algorithms, software, and hardware systems do not match the categories, often blending approaches.
- No attempt to cover all models and aspects of parallel computing. For example, quantum computing not included.
Parallel Computing Thesaurus

Parallel Computing  Solving a task by simultaneous use of multiple processors, all components of a unified architecture.

Embarrassingly Parallel  Solving many similar, but independent, tasks. E.g., parameter sweeps.

Symmetric Multiprocessing (SMP)  Multiple processors sharing a single address space, OS instance, and all resources.

Multi-core Processors  Multiple processors (cores) on a single chip. Aka many-core. Some are heterogeneous CPU/GPU combinations.

Cluster Computing  Hierarchical combination of commodity units (processors or SMPs) to build parallel system.
Supercomputer  The fastest, biggest machines designed to solve large problems. Historically vector computers, but now are parallel, perhaps with GPUs.

High Performance Computing  Solving large problems via supercomputers + fast networks and storage + visualization.

Pipelining (streaming)  Breaking a task into steps performed by different units, with inputs streaming through, much like an assembly line.
Who Uses Supercomputers?

- Energy
- Automotive
- Finance
- Database
- Telecomm
- Weather & Climate
- Aerospace
Top 500 Performance

The Top 500 performance as of June. The newest list has just been announced: [http://www.top500.org](http://www.top500.org)
A greatly simplified model, based on parallelizing crash simulation for Ford Motor Company. Simulations save significant money and time compared to testing real cars.

This example illustrates various phenomena which are common to many simulations and other large-scale applications.

This material is also relevant for developing new codes.
Finite Element Representation

- Car is modeled by a triangulated surface (the elements).

- The simulation consists of modeling the movement of the elements during each time step, incorporating the forces on them to determine their new position.

- In each time step, the movement of each element depends on its interaction with the other elements that it is physically adjacent to.
The Car of the Future
Basic Serial Crash Simulation

1. For all elements
2. Read State(element), Properties(element), Neighbor_list(element)
3. For time=1 to end_of_simulation
4. For element = 1 to num_elements
5. Compute State(element) for next time step, based on previous state of element and its neighbors, and on properties of element

Periodically State is stored on disk for later visualization.
Simple approach to parallelization

Parallel computer based on PC-like processors linked with a fast network, where processors communicate via messages. *Distributed memory* or *message-passing*.

Distribute elements to processors, each processor updates the positions of the elements it contains: *owner computes*.

All machines run the same program: *SPMD*, single program multiple data.

*SPMD is the dominant form of parallel computing.*
A Distributed Car
Basic Parallel Version

Concurrently for all processors P

1 For all elements assigned to P
2 Read State(element), Properties(element), Neighbor-list(element)
3 For time=1 to end-of-simulation
4 For element = 1 to num-elements-in-P
5 Compute State(element) for next time step, based on previous state of element and its neighbors, and on properties of element
Most parallel code the same as, or similar to, serial code, reducing parallel development and life-cycle costs, and helping keep parallel and serial versions compatible.

High-level structure same as serial version: a sequence of steps. The sequence is a serial construct, but steps are performed in parallel.

Use incremental parallelization, constantly checking versus test cases.

Life-cycle costs are often overlooked until it is too late!
Some Basic Questions: Allocation?

How are elements assigned to processors?

Typically element assignment determined by serial preprocessing, using domain decomposition approaches (load-balancing) described later.
Separation?

How does processor keep track of adjacency info for neighbors in other processors?

- Use *ghost cells* (halo) to copy remote neighbors, add translation table to keep track of their location and which local elements copied elsewhere.
Update?

How does a processor use State(neighbor) when it does not contain the neighbor element?

Could request state information from processor containing the neighbor. However, more efficient if that processor sends it.
An important component of effective parallel computing is determining whether the program is performing well. If it is not running efficiently, or cannot be scaled to the target number of processors, then one needs to determine the causes of the problem and develop better approaches.
Definitions

For a given problem A, let

$\text{SerTime}(n) =$ Time of best serial algorithm to solve A for input of size $n$.

$\text{ParTime}(n,p) =$ Time of the parallel algorithm+architecture to solve A for input of size $n$, using $p$ processors.

Note that $\text{SerTime}(n) \leq \text{ParTime}(n,1)$.

**Speedup:** $\text{SerTime}(n) / \text{ParTime}(n,p)$

**Work (cost):** $p \cdot \text{ParTime}(n,p)$

**Efficiency:** $\text{SerTime}(n) / [p \cdot \text{ParTime}(n,p)]$
Expect:

\[ 0 < \text{Speedup} \leq p \]
\[ \text{Serial Work} \leq \text{Parallel Work} < \infty \]
\[ 0 < \text{Efficiency} \leq 1 \]

*Linear speedup*: speedup = \( p \).

Sometimes called *perfect speedup*
Always involves some restriction on relationship of \( p \) and \( n \),
e.g., \( p \leq n \), or \( p = \sqrt{n} \).
Observed Speedup

- perfect
- common
- occasional

processors

speedup

speedup
Superlinear Speedup

Very rare. Some reasons for speedup $> p$ (efficiency $> 1$)

- Parallel computer has $p$ times as much RAM so higher fraction of program memory in RAM instead of disk. *An important reason for using parallel computers*

- In developing parallel program a better algorithm was discovered, older serial algorithm was not best possible. *A useful side-effect of parallelization*
Amdahl’s Law

Amdahl [1967]: Let $f$ be fraction of time spent on operations that are performed serially. Then for $p$ processors,

$$ParTime(p) \geq SerTime(p) \cdot \left[ f + \frac{1 - f}{p} \right]$$

(Assumes perfect parallelization of (1-f) part of program)

$$Speedup(p) \leq \frac{1}{f + (1 - f)/p}$$

Thus no matter how many processors are used:

$$Speedup \leq \frac{1}{f}$$
Unfortunately, typically $f$ was 10 – 20%, i.e., in best possible parallization, speedup could be 5 – 10.

Compounding the difficulty:

If maximal possible speedup is $S$, then $S$ processors run, at best, at about 50% efficiency.

If $S = 1/f$ then $\text{ParTime}(S) = \text{SerTime} \cdot [f + (1-f)/S] = \text{SerTime} \cdot [2f - f^2]$

$\approx \text{SerTime} \cdot 2f = \text{SerTime} \cdot 2/S$
Maximal Possible Speedup

The graph shows the speedup as a function of the number of processors for different values of the parameter $f$. The x-axis represents the number of processors, ranging from 1 to 1024, and the y-axis represents speedup, ranging from 1 to 1024.

For $f=0.1$, the speedup increases linearly with the number of processors. For $f=0.01$, the speedup also increases linearly but at a slower rate. For $f=0.001$, the speedup increases at an even slower rate, approaching a constant value as the number of processors increases.

This graph illustrates the maximal possible speedup achievable with a given number of processors and parameter $f$. It highlights the trade-off between the number of processors and the speedup, which is crucial for optimizing parallel computing applications.
Maximal Possible Efficiency

![Graph showing efficiency versus processors for different values of f (0.001, 0.01, 0.1)].

- For f = 0.001, the efficiency decreases rapidly with an increasing number of processors.
- For f = 0.01, the efficiency decreases more gradually compared to f = 0.001.
- For f = 0.1, the efficiency remains relatively constant across a wide range of processors.

Stout and Jablonowski – p. 32/268
Parallelization usually adds communication.

For Crash: ghost cells sent every time step and periodic global communication to check if parts are colliding.
Amdahl was a Pessimist

Amdahl convinced many that general-purpose parallel computing was not viable. Fortunately, we can skirt the law.

**Algorithm:** New algorithms with much smaller values of \( f \). *Necessity is the mother of invention.*

**Memory hierarchy:** More time spent in RAM than disk.

**Scaling:** Usually time spent in serial portion of code is a decreasing fraction of the total time as problem size increases.
Common Program Structure

Often serial part grows with $n$ much slower than total time.

Serial, grows slowly with $n$

Parallelizable loop, grows with $n$

Serial, fixed time

Parallelizable loop within loop, grows very rapidly with $n$

Serial, grows slowly with $n$

I.e., Amdahl’s “$f$” decreases as $n$ increases
Scaling

- Can often use this to exploit large parallel machines by scaling the problems to larger n

- Fix the amount of data per processor: weak scaling
  - Efficiency can remain high if communication does not increase excessively
  - Warning: efficiency improves, but parallel time will increase if SerTime(n) superlinear (ω(n)).

- Amdahl considered strong scaling.

- When you see or present scaling results, make sure you know which scaling is being used.
Scalability

Linear speedup is rare, due to communication overhead, load imbalance, algorithm/architecture mismatch, etc.

However, for most users, the important question is:

*Have I achieved acceptable performance on my software/hardware system for a suitable range of data and machine sizes?*
These classifications provide ways to think about problems and their solution.

The classifications were originally in terms of hardware, but there are natural software analogues.

Many systems blend approaches, and do not exactly correspond to the classifications.
Flynn’s Instruction/Data Taxonomy

[Flynn, 1966] At any point in time can have

\[
\begin{align*}
\{ \text{S} \} & \quad \text{I} \quad \{ \text{S} \} \\
\{ \text{M} \} & \quad \text{D}
\end{align*}
\]

**SI** Single Instruction: All processors execute the same instruction.

**MI** Multiple Instruction: Different processors may be executing different instructions.

**SD** Single Data: All processors are operating on the same data.

**MD** Multiple Data: Different processors may be operating on different data.
- SISD: standard serial computer and program.
- MISD: very rare — some extreme fault-tolerance schemes, using different computers and programs to operate on same input data, are of this type.
- MIMD: Almost all parallel computers are of this form. This is the primary focus of this class.
- SIMD: there used to be large SIMD systems (Thinking Machines’ Connection Machine). Now:
  - GPUs have SIMT units (T="threads")
  - Intel has Streaming SIMD Extension (SSE) instructions
A Conceptual View of SIMD

Controller, with program

Instruction

Processors, with data
**SIMD Software**

*Data parallel*: do the same thing to all elements of a structure (e.g., many matrix algorithms).

Easiest to write and understand.

Unfortunately, difficult to apply to complex problems (as were the SIMD machines).

*SPMD, Single Program Multiple Data*: All processors use the same program.

A coarse-grained SIMD approach to programming for MIMD systems.

Dominant form of parallel programming.
Shared vs. Distributed Memory

**SHARED MEMORY**
- Memory
- Network
- Processor + Cache

**DISTRIBUTED MEMORY**
- Network
- Processor + Cache + Memory

Stout and Jablonowski – p. 43/268
Distributed Memory (DM)

- If processor \( A \) needs data in processor \( B \), then \( B \) must send a message to \( A \) containing the data. Thus DM systems also known as *message passing* systems.

- Advantages:
  - Memory is scalable with number of processors
  - Each processor has rapid access to its own memory
  - *Cost effective*: can use commodity parts

- Disadvantages:
  - Programmer is responsible for many of the details of the communication, easy to make mistakes.
  - May be difficult to distribute the data structures
Shared Memory (SM)

- Global memory space, accessible by all processors
- Processors may have local copies of some global memory.
- Consistency of copies is usually maintained by hardware (cache coherency)

Advantages:
- Global address space is user-friendly
- Data sharing between tasks is fast
Disadvantages

- Scalability problematic, shared memory - to - CPU path may be a bottleneck
- Programmer is responsible for correct synchronization
- Large systems need some special-purpose components. This adds cost.
SM/DM Hardware/Software

Problem: programmers prefer SM, accountants prefer DM. 

Software and hardware models need not match

SM software on DM hardware:

- Global address space (GAS) languages use software to simulate SM. Include Unified Parallel C (UPC) and Co-Array Fortran (CAF), part of Fortran 2008 standard.
- However, GAS efficiency is often problematic because they are trying to mask significant latency.

DM software on SM hardware:

- MPI (discussed later) is for DM programming, but always available on SM systems.
- SM message passing: simple, fast, memory copying.
Communication Network

There are many networks, but for the user the differences usually minor. Two main classes that do have some impact:

**Bus:** Processors (and memory) connected to a common bus or busses, much like a local Ethernet.

- Not very scalable due to contention.

Switching Network: Similar to a telephone system.
- Many messages can be transmitted simultaneously.
- These networks vary widely in price, based on capabilities. Many are proprietary.

[YouTube Video](http://www.youtube.com/embed/ystkKXzt9Wk?rel=0) Created by Aaron Koblin
Example: Symmetric Multiprocessors

- Shared memory system, processors share work.
- When a processor reads or writes to RAM, data transported over a bus, local copy in processor cache.
- Hardware may ensure that different caches don’t contain different values for the same memory locations (cache coherency). Easier on bus-based systems than on more general interconnection networks.
- Because all processors use the same memory bus, there is limited scalability due to bus contention.
- Multicore processors themselves can be SMPs.
Memory Hierarchy

*von Neumann bottleneck*: processor much faster than memory, sits idle waiting for data. Unfortunately, faster memory higher $/byte, physics imposes size constraints.

To ameliorate latency, data moved between levels in blocks (cache lines, pages). For efficiency: *use entire block while resident in the faster memory*
Parallel Computing Has Hurdles

The parallel programmer faces many hurdles. You might find some more difficult than others.

news.bbc.co.uk
Serialization Causes Bottlenecks

There are several causes of this, and we’ll show how to identify and avoid many of them.
Workload is Not Evenly Distributed

- We’ll show several ways to load balance various types of problems.
Debugging is Harder

- We’ll show approaches to developing the software, and debugging tools.

Where’s Waldo’s Error?

From howardforums.com
Current Approach Doesn’t Parallelize

Often the most difficult hurdle.

- Identify the inherent parallelism in the application.
- Think out of the box and explore new approaches.
On distributed memory systems, also called message passing systems, communication is often an important aspect of performance and correctness.

- Messages are like handshakes.
- They need two partners: a sender and receiver.

Source: thetrafficmagic.com/Handshake.jpg
Communication Speed

On most distributed memory systems, messages are relatively slow, with startup (latency) times taking thousands of cycles (and far more for many clusters).

Typically, once the message has started, the additional time per byte (bandwidth) is relatively small.
For example, a 2.6 GHz Intel Xeon E5-2670 (Sandy Bridge) processor with Advanced Vector Extensions (AVX),

**Best case** MPI messages (theoretical peak):

- processor speed: 2600 cycles per microsecond ($\mu$sec),
  8 flops/cycle: 20800 flops per $\mu$sec.

- MPI message latency, caused by software:
  \[
  \approx 2.5 \ \mu \text{sec} = 52000 \text{ flops}
  \]

- message bandwidth, usually limited by hardware:
  \[
  \approx 13600 \text{ bytes per } \mu \text{sec} = 1.53 \text{ flops/byte}
  \]

*Your performance may vary!*
Reducing Latency

Reducing the effect of high latency often important for performance. Some useful approaches:

- Reduce the number of messages by mapping communicating entities onto the same processor.
- Combine messages having the same sender and destination.
- If processor $P$ has data needed by processor $Q$, have $P$ send to $Q$, rather than $Q$ first requesting it. $P$ should send as soon as data ready, $Q$ should read as late as possible to increase probability data has arrived.

Send Early, Receive Late, Don’t Ask but Tell.
Reduce the Network Traffic

- In a congested traffic system, significant reduction of vehicles (messages) can greatly improve the delay (latency) seen by all vehicles (processors).
- Bundle the messages for each sender-receiver pair.
- This resembles carpooling:
Deadlock

If messages *blocking*, i.e., if processor can’t proceed until the message is finished, then can reach *deadlock*, where no processor can proceed.

Example: Processor $A$ sends message to $B$ while $B$ sends to $A$. If blocking sends, neither finishes until the other finishes receiving, but neither starts receiving until send finished.

This can be avoided by $A$ doing send then receive, while $B$ does receive then send. However, often difficult to coordinate when there are many processors.
Blocking versus Non-Blocking

Often easiest to prevent deadlock by non-blocking communication, where processor can send and proceed before receive is finished.

However, requires receiver buffer space which may fill (and hence cause blocking), and extra copying of messages, reducing performance.
Another Example of a Deadlock
Message Passing Interface — MPI

An important communication standard. We will show some snippets of MPI to illustrate some of the issues, but MPI is a major topic that we cannot address in detail. Fortunately, many programs need only a few MPI features. There are many implementations of MPI:

MPICH homepage  http://www.mpich.org
Open MPI homepage  http://www.open-mpi.org/
Message Passing Interface Forum (official MPI standards documents)  http://www.mpi-forum.org/
Some Reasons for Using MPI

- Standardized, with process to keep it evolving.
- Available on almost all parallel systems (free MPICH, Open MPI used on many clusters), with interfaces for C/C++ and Fortran.
- Supplies many communication variations and optimized functions for a wide range of needs.
- Works both on distributed memory (DM) and shared memory (SM) hardware architectures
- Supports large program development and integration of multiple modules.
- Many powerful packages and tools based on MPI.

Stout and Jablonowski – p. 66/268
While MPI large (> 100 functions), usually need very few functions (6-10), giving gentle learning curve.

Various training materials, tools and aids for MPI.

- Good introductory MPI tutorial
  
  https://computing.llnl.gov/tutorials/mpi/

- Basic and advanced MPI tutorials, e.g. on I/O and one-sided communication
  
  http://www-unix.mcs.anl.gov/mpi/tutorial/
  http://www.citutor.org/browse.php
While MPI large (> 100 functions), usually need very few functions (6-10), giving gentle learning curve.

Various training materials, tools and aids for MPI.

- Good introductory MPI tutorial
  [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)

- Basic and advanced MPI tutorials, e.g. on I/O and one-sided communication

Writing MPI-based parallel codes helps preserve your investment as systems change.
The overwhelmingly most frequently used MPI commands are variants of

- `MPI_SEND()` to send data, and
- `MPI_RECV()` to receive it.

These function very much like write & read statements.

*Point-to-point communication*

- `MPI_SEND()` and `MPI_RECV()` are *blocking* operations.
- Blocking communication can be unsafe and may lead to deadlocks.
Blocking MPI Communication

- *MPI_SEND()* does not complete until the communication buffer is empty
- *MPI_RECV()* does not complete until the communication buffer is full

Send-recv handshake works for small messages, but might fail for large messages

- Allowable size of the message depends on MPI implementation (buffer sizes)
- Even if it works, the data usually get copied into a memory buffer
- Copies are slow (avoid), poor performance
Non-Blocking MPI Communication

Better solution: use non-blocking operations

- `MPI_Isend()`
- `MPI_Irecv()`
- `MPI_Wait()`

The user can also check for the data at a later stage in the program without waiting:

- `MPI_Test()`

Non-blocking operations boost the performance.

Other non-blocking send and receive operations available.

Possible overlap of communication with computation.

However, few systems can provide the overlap.
Near the beginning of the program, include

```c
#include "mpi.h"
MPI_Init(&argc, &argv)
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
MPI_Comm_size(MPI_COMM_WORLD, &num_processors)
```

These help each processor determine its role in the overall scheme.

There is `MPI_Finalize()` at the end.

These 4 MPI functions, together with MPI send and receive operations, are already sufficient for simple applications.
Each processor sends value to proc. 0, which adds them.
initialize

if (my_rank == 0){
    sum = 0.0;
    for (source=1; source<num_procs; source++){
        MPI_RECV(&value, 1, MPI_FLOAT, source, tag,
                  MPI_COMM_WORLD, &status);
        sum += value;
    }
} else {
    MPI_SEND(&value, 1, MPI_FLOAT, 0, tag,
              MPI_COMM_WORLD);
}

finalize
Improving Performance

In the initial version, processor 0 received the messages in processor order. However, if processor 1 delayed sending its message, then processor 0 would also be delayed.

For a more efficient version: modify MPI_RECV to

```
MPI_Recv(&value, 1, MPI_FLOAT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
```

Now processor 0 can start processing messages as soon as any arrives.
Reduction Operations

Operations such as summing are common, combining data from every processor into a single value. These reduction operations are so important that MPI provides direct support for them, and parallelizing compilers recognize them and generate efficient code.

Could replace all communication with

```
MPI_REDUCE(&value, &sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Examples of reduction operations:

- MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD
- MPI_LAND (logical and), MPI_LOR (logical or)
Collective Communication

The opposite of reduction is broadcast, one processor sends to all others:

\[ \text{MPI\_Bcast(buf, count, type, root, comm);} \]

Reduction, broadcast, and others are collective communication operations, the next most frequently invoked MPI routines after send and receive.

MPI collective communication routines improve clarity, run faster, and reduce chance of programmer error. Other examples are

\[ \text{MPI\_Scatter, MPI\_Gather, MPI\_Allgather, MPI\_Alltoall} \]
Collective Communication

Broadcast

P0
P1
P2
P3

A

A
A
A

P0
P1
P2
P3

Scatter

P0
P1
P2
P3

A A A A
B C D
C
D

Stout and Jablonowski – p. 77/268
Collective Communication

All gather

P0 A
P1 B
P2 C
P3 D

A B C D
P0
A B C D
P1
A B C D
P2
A B C D
P3

All to all

P0 A0 A1 A2 A3
P1 B0 B1 B2 B3
P2 C0 C1 C2 C3
P3 D0 D1 D2 D3

A0 B0 C0 D0 P0
A1 B1 C1 D1 P1
A2 B2 C2 D2 P2
A3 B3 C3 D3 P3
MPI Synchronization

Synchronization is provided

- implicitly by
  - Blocking communication
  - Collective communication
    (new: non-blocking versions available in MPI-3)

- explicitly by
  - `MPI_Wait, MPI_Waitany` operations for non-blocking communication:
    May be used to synchronize a few or all processors
  - `MPI_Barrier` statement:
    Blocks until all MPI processes have reached barrier

Avoid synchronizations as much as possible to boost performance.

Stout and Jablonowski – p. 79/268
MPI Datatypes

- Predefined basic datatypes, corresponding to the underlying programming language, examples are:
  - Fortran
    - MPI_INTEGER
    - MPI_REAL, MPI_DOUBLE_PRECISION
  - C
    - MPI_INT
    - MPI_FLOAT, MPI_DOUBLE
- Derived data types:
  - Vector: data separated by constant stride
  - Contiguous: vector with stride 1
  - Struct: general mixed types (e.g. for C struct)
  - Indexed: Array of indices
MPI Datatype: Vector

Consider a block of memory (e.g. a matrix with integer numbers):

```
  1  5  9  13  17  21
  2  6 10  14  18  22
  3  7 11  15  19  23
  4  8 12  16  20  24
```

To specify the gray row (in Fortran order), use

```
MPI_Type_vector( count, blocklen, stride, old_datatype, new_datatype, ierr)
MPI_Type_commit (new_datatype, ierr)
```
**MPI Datatype: Vector**

- In the example, we get (in Fortran notation)
  
  ```fortran
  MPI_Type_vector( 6, 1, 4, MPI_INTEGER, 
                  my_vector, ierr)
  MPI_Type_commit (my_vector, ierr)
  ```

- The new datatype `my_vector` is a vector that contains 6 blocks, each of 1 integer number, with a stride of 4 integers between blocks.

- Here, we introduce the Fortran notation of the MPI routines (with additional error flag "ierr").

- Fortran, C and C++ notations are very similar.
Some Additional MPI Features

- Procedures for creating virtual topologies, e.g., indexing processors as a 2-dimensional grid.

- User-created communicators (e.g., replace MPI_COMM_WORLD), useful for selective collective communication (e.g., summing along rows of a matrix), incorporating software developed separately.

- Support for heterogeneous systems, MPI converts basic datatypes.

- Additional user-specified derived datatypes

- Parallel I/O, critical for scalability of I/O intensive problems
One-sided Communication

- Essentially “put” and “get” operations that can greatly improve efficiency on some codes.
- Conceptually the same as directly accessing remote memory.

However, they are risky and can easily introduce race conditions.
The MPI-3 standard was finalized in September 2012. New features in MPI-3 include:

**Non-blocking collective communication** Allows overlap of computations and communication. E.g. “broadcast”

```c
MPI_Isbcast(buf, count, type, root, comm,&request);
... // compute
MPI_Wait (&request, &status);
```

**Remote Memory Access (RMA)** Improves global memory access, similar to “Partitioned Global Address Space” (PGAS) languages like “Co-array Fortran/Fortran 2008” and “Unified Parallel C” (UPC).

**Shared Memory Windows** Allows sharing of single objects (e.g. arrays or data structures)
The MPI standard includes
- point-to-point message-passing
- collective communication
- one-sided communication and PGAS memory model
- parallel I/O routines
- group and communicator concepts
- process topologies (e.g. graphs)
- environmental management (e.g. timers, error handling)
- process creation and management
- profiling interface
Here we address the question of how one goes about subdividing the computational domain among the processors. We introduce the basic techniques that are applicable to most programs, with some more advanced techniques appearing later.
Which processor is the most important for parallel performance?
Domain and Functional Decomposition

**Domain decomposition:** Partition a (perhaps conceptual) space. Different processors do similar work on different pieces (quilting bee, teaching assistants for discussion sections, etc.)

**Functional decomposition:** Different processors work on different types of tasks (workers on an assembly line, sub-contractors on a project, etc.)

Functional decomposition rarely scales to many processors, so we’ll concentrate on domain decomposition.
Dependency Analysis

There is a *dependency* between $A$ and $B$ if value of $B$ depends upon $A$. $B$ cannot be computed before $A$.

*Dependencies control parallelization options.*
Almost always

- *Time* or time-like variables and operations (signals, non-commutative operations, etc.) cannot be parallelized.

- *Space* or space-like variables and operations (names, objects, etc.) can be parallelized.

Some operations can have both time-like and space-like properties. E.g., ATM transactions are usually to independent accounts (space-like), but ones to the same account must be done in order (time-like).
Loops often introduce real, or apparent, dependencies.

do i=1,n
    V[i]=V[i] − 2*V[i−1]
enddo

**Backward dependency**: cannot be parallelized because each value depends upon value from previous iteration.

\[ V \]

Must be computed before

This can be computed
do i=1,n
  V[i]=V[i] − 2*V[i+1]
enddo
Forward Dependency

do i=1,n
   V[i]=V[i] − 2*V[i+1]
enddo

V

Must be computed before This is updated

Race

Solution: V_copy=V
do i=1,n
   V[i]=V_copy[i] − 2*V_copy[i+1]
enddo

Some parallelizing compilers do this automatically.
A few things appear to be serial but can be parallelized.

**Reduction**

\[
x = 0 \\
\text{for } i = 0, n-1 \\
\quad x = x + a[i] \\
\text{end for}
\]

**Scan or Parallel Prefix**

\[
y[0] = a[0] \\
\text{for } i = 1, n-1 \\
\quad y[i] = y[i-1] + a[i] \\
\text{end for}
\]
Parallelized Reduction Operations

- Reduction and scan operations are very common.

- They are recognized by parallelizing compilers and implemented in MPI (MPI_REDUCE, MPI_SCAN) and OpenMP (REDUCTION clause).

```
+    +  +  +
  +    +  +  +
```

Stout and Jablonowski – p. 95/268
Load-Balancing Variety

Many different types of load-balancing problems:
- static or dynamic,
- parameterized or data dependent,
- homogeneous or inhomogeneous,
- low or high dimensional,
- graph oriented, geometric, lexicographic, etc.

Because of this diversity, need many different approaches and tools.
Static Decompositions

We start with static decompositions of the work, with dynamic decompositions discussed later.

Often just evenly dividing space among the processors yields acceptable load balance, with acceptable performance if communication minimized.

This approach works even if the objects have varying computational requirements, as long as there are enough objects so that the worst processor is likely to be close to the average (law of large numbers).
Which Matrix Decomposition is Best?

Suppose work at each position only depends on value there and nearby ones, equivalent work at each position.

Minimizing Boundary

Minimizing Number of Neighbors
Matrix Decomposition Analysis

- Computation proportional to area so both load balanced.

- Squares minimize bytes communicated (parallelization overhead), so is generally better.

- *However:* Recall, there is significant overhead in starting a message, especially on clusters, so far smaller matrices may need to concentrate on number, not size, of messages, i.e., use strips.
Local vs. Global Arrays

Serial Array

0 \cdots 9 10 \cdots 19 20 \cdots 29

Distributed Array

-1 0 \cdots 9 10

Processor 0

-1 0 \cdots 9 10

Processor 1

ghost
(if needed)

-1 0 \cdots 9 10

Processor 2
If serial has matrix $A[0:n-1]$, and there are $p$ DM processors, with ranks $0 \ldots p-1$

- each processor has matrix $A[0:n_{local}-1]$, where $n_{local} = \lceil n/p \rceil$ (round up)

- $A[i]$ on processor $q$ corresponds to $A[i+q*n_{local}]$ in the serial array (except for $i \geq n-(p-1)*\lceil n/p \rceil$ on $q=p-1$)

- if use $A[i+1]$ and $A[i-1]$ in calculation of $A[i]$, ($i \neq 0, n-1$), then use $A[-1:n_{local}]$ to add ghost cells

Can also have $A[i]$ on proc $q$ correspond to $A[i+\lceil q \cdot n/p \rceil]$ in serial array
### MPI Rank vs. 2-D Indices

For \( \text{MPI\_COMM\_RANK} = i \) and \( \text{MPI\_COMM\_SIZE} = p \)

\[
\text{my\_row} = \left\lfloor \frac{i}{\sqrt{p}} \right\rfloor \quad \text{and} \quad \text{my\_col} = i - \text{my\_row} \times \sqrt{p}
\]

<table>
<thead>
<tr>
<th>to send to logical</th>
<th>send to MPI_COMM_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right (my_row, my_col +1)</td>
<td>i + 1</td>
</tr>
<tr>
<td>Left (my_row, my_col -1)</td>
<td>i − 1</td>
</tr>
<tr>
<td>Up (my_row -1, my_col)</td>
<td>i − \sqrt{p}</td>
</tr>
<tr>
<td>Down (my_row +1, my_col)</td>
<td>i + \sqrt{p}</td>
</tr>
</tbody>
</table>

MPI “virtual topologies” can do this for you.
Graph Decompositions

Very general graph decomposition techniques can be used when communication patterns are less regular.

- Objects (calculations) represented as vertices (with weights if calculation requirements are uneven).
- Communication represented as edges (with weights if communication requirements are uneven).

Goals:

1. Assign vertices to processors to evenly distribute the number/weight of vertices: *balance computation*

2. Minimize and balance the number/weight of edges between processors: *minimize communication*
What is the Best Decomposition?

Numbers indicate work, want to use 4 processors.
What is the Best Decomposition?

Numbers indicate work, want to use 4 processors.

Processors:
Graph Decomposition Tools

- Optimal graph decomposition is NP-hard.
- Fortunately, several heuristics work well
- High-quality decomposition tools, such as Metis, available.
  [http://glaros.dtc.umn.edu/gkhome/metis/metis/overview](http://glaros.dtc.umn.edu/gkhome/metis/metis/overview)
- Parallel version, ParMetis, also available.
Using Serial Decomposition Tool

Problem

Convert

Metis

Convert

Parallel Program

Perl, Python, etc.
Where Do Weights Come From?

If weights are static and objects of the same type have about the same requirements, and if types are known in advance, then:

- Sometimes all the same.
- Sometimes easy to deduce a priori.
- May use simple measurements on small test cases.
- May use statistical curve fitting on sample problems.

If types aren’t known in advance, this won’t be useful.
Static Geometric Decompositions

When the objects have an underlying geometrical basis, such as the finite elements representing surfaces of car parts, or polygons representing census blocks in a geographical information system, then the geometry can often be exploited

*if communication predominately involves nearby objects.*

Geometric decompositions can be based on recursive bisectioning, quad- or oct-trees, ham sandwich theorems, space-filling curves, etc., and can incorporate weights.

Warning: geometric approaches not nearly as useful on high-dimensional data.
Space-Filling Curves

The best general-purpose geometric load-balancing comes from space-filling curves.

The Hilbert Space-Filling Curve
The best general-purpose geometric load-balancing comes from space-filling curves.

The Hilbert Space-Filling Curve
The best general-purpose geometric load-balancing comes from space-filling curves.

### The Hilbert Space-Filling Curve

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 63| 62| 49| 48| 47| 46| 45| 44| 43| 42| 59| 58| 57| 55| 54| 53| 52| 33| 34| 35| 36| 37| 29| 28|
| 60| 61| 50| 51| 46| 45| 40| 41| 32| 31| 27| 26| 30| 29| 24| 25| 23| 22| 17| 18| 20| 21| 14| 15| 16|

For an implementation, see the references.
Using A Space-Filling Curve

Letters represent work, boldface twice as much work.
Step 1: Determine Space-Filling Coordinates

A  B  C  D
E  F
G  H  I  J  K  L  M
N  O  P
Q  R  S
T  U  V  W
X  Y
Z

A  B  C  D
E  F
G  H  I  J  K  L  M
N  O  P
Q  R  S
T  U  V  W
X  Y
Z
Step 2: Sort by Space-Filling Coordinates
Step 3: Divide Work Evenly Based on Sorted Order

A   B   C   D
63   49   48   47
E   F
50   40
G   H   I   J   K   L   M
59   56   55   52   34   39   38
N   O   P
58   53   36
Q   R   S
6    31   27
T   U   V   W
8    11   24   25
X   Y
2    23
Z
19
Z- Ordering

Aka Morton or shuffled bit ordering. For 2-D, point \((x_2 x_1 x_0, y_2 y_1 y_0)\) mapped to \(y_2 x_2 y_1 x_1 y_0 x_0\)

For 3-D, \((x_k \ldots x_1 x_0, y_k \ldots y_1 y_0, z_k \ldots z_1 z_0)\) \(\rightarrow\) \(z_k y_k x_k \ldots z_1 y_1 x_1 z_0 y_0 x_0\)
Which SFC to Use?

- Both extend to arbitrary dimensions.
- Both give regions with boundary (communications) within constant factor of optimal.
- Hilbert ordering assigns only 1 contiguous region to a processor, Z ordering may assign 2.
- Z slightly easier to compute than Hilbert.

In practice, little difference in performance.
Parallel programming on shared memory (SM) machines has always been important in high performance computing.

- All processors can access all the memory in the parallel system (access time can be different).
- In the past: Utilization of such platforms has never been straightforward for the programmer.
- Vendor-specific solutions via directive-based compiler extensions dominated until the mid 90’s.
Parallelization Techniques: OpenMP

- Since 1997: OpenMP is the industry standard for shared memory programming.
- In 2013: The OpenMP Version 4.0 specification was released (new feature: accelerator directives).
- OpenMP is an Application Program Interface (API): directs multi-threaded shared memory parallelism ⇒ *thread based parallelism*
- **Explicit** (not automatic) programming model: the programmer has full control over the parallelization, compiler interprets parallel constructs.
- Based on a combination of compiler directives, library routines and environment variables.
- OpenMP uses the fork-join model of parallel execution.
OpenMP can be interpreted by most commercial Fortran and C/C++ compilers, supports all shared-memory architectures including Unix and Windows platforms, and hence should be your programming system of choice for shared memory platforms.

OpenMP home page and recommended online tutorial:

http://www.openmp.org

https://computing.llnl.gov/tutorials/openMP/
Goals of OpenMP

- **Standardization**: standard among all shared memory architectures and hardware platforms
- **Lean**: simple and limited set of compiler directives for shared memory machines. Often significant parallelism by using just 4-6 directives.
- **Ease of use**: supports incremental parallelization of a serial program, unlike MPI which typically requires an all or nothing approach.
- **Portability**: supports Fortran, C and C++
OpenMP: 3 Building Blocks

Compiler directives (imbedded in user code) for
- parallel regions (PARALLEL)
- parallel loops (PARALLEL DO)
- parallel workshare (PARALLEL WORKSHARE)
- parallel sections (PARALLEL SECTIONS)
- parallel tasks (PARALLEL TASK)
- sections to be done by only one processor (SINGLE)
- synchronization (BARRIER, CRITICAL, ATOMIC, ...)
- data structures (PRIVATE, SHARED, REDUCTION)

Run-time library routines (called in the user code) like
OMP_SET_NUM_THREADS,
OMP_GET_NUM_THREADS, etc.

UNIX Environment variables (set before program execution) like OMP_NUM_THREADS, etc.
Parallel execution is achieved by generating *threads* which are executed in parallel (multi-threaded parallelism):
OpenMP: The Fork-Join Model

- Master thread executes sequentially until the first parallel region is encountered.

**FORK:** The master thread creates a team of threads which are executed in parallel.

**JOIN:** When the team members complete the work, they synchronize and terminate. The master thread continues sequentially.

- Number of threads is independent of the number of processors.

- Quiz: What happens if
  - # threads or tasks > # processors
  - # threads or tasks < # processors

Stout and Jablonowski – p. 122/268
OpenMP: Work-sharing Constructs

- **DO/for loops:** type of “data parallelism”
- **WORKSHARE:** (Fortran only) breaks work into independent units (data parallelism)
- **SECTION:** breaks work into independent sections that are executed concurrently by a thread (“functional parallelism”), units of work are statically defined
- **TASK:** breaks work into independent tasks that are executed asynchronously in the form of dynamically generated units of work (“irregular parallelism”),
- **SINGLE:** serializes a section of the code. Useful for sections of the code, that are not threadsafe (I/O).

OpenMP recognizes compiler directives that start with

- !$OMP (in Fortran)
- #pragma omp (in C/C++)

Stout and Jablonowski – p. 123/268
OpenMP: Work-sharing Constructs

- **DO/for loop WORKSHARE**
- **SECTIONS**
- **SINGLE**

No barrier upon entry to these constructs, but implied barrier (synchronization) at the end of each ⇒ functionality of the OpenMP directive !$OMP BARRIER
OpenMP Barrier

- Barriers maybe needed for correctness
- Synchronization degrades performance, avoid if possible
Parallel Loops (1)

⇒ in Fortran notation

Parallel loop is embedded in a parallel region

```fortran
!$OMP PARALLEL
!$OMP DO
DO i = 1, n
    a(i) = b(i) + c(i)
END DO
!$OMP END DO
!$OMP END PARALLEL
```
Parallel Loops (2)

- Each thread executes a part of the loop.
- By default, the work is evenly and continuously divided among the threads ⇒ e.g. 2 threads:
  - thread 1 works on \( i = 1 \ldots \frac{n}{2} \)
  - thread 2 works on \( i = \left(\frac{n}{2} + 1\right) \ldots n \)
- The work (number of iterations) is statically assigned to the threads upon entry to the loop.
- Number of iterations cannot be changed during the execution.
- Implicit synchronization at the end, unless “NOWAIT” clause is specified.
- Highly efficient, low overhead.
Parallel Workshare

Workshare embedded in a parallel region:
The block of work must be simple, e.g. can only contain
data assignments and a few other constructs like FORALL
or WHERE statements.

Arrays aa, bb, cc, dd and scalar “first” are shared:

!$OMP PARALLEL SHARED (aa,bb,cc,dd,first)

!$OMP WORKSHARE

cc = aa * bb
dd = aa + bb
first = cc(1,1) + dd(1,1)

!$OMP END WORKSHARE NOWAIT

!$OMP END PARALLEL
Parallel Sections (1)

⇒ in Fortran notation

!$OMP PARALLEL SECTIONS

 !$OMP SECTION
 DO i = 1, n
   a(i) = b(i) + c(i)
 END DO

 !$OMP SECTION
 DO i = 1, k
   d(i) = e(i) + e(i-1)
 END DO

 !$OMP END PARALLEL SECTIONS
Parallel Sections (2)

- The two independent sections can be executed concurrently by two threads.
- Units of work are statically defined at compile time.
- Each parallel section is assigned to a specific thread, executes work from start to finish.
- Thread cannot suspend the work.
- Implicit synchronization unless “NOWAIT” clause is specified.
- Nested parallel sections are possible, but can be costly due to high overhead of parallel region creation.
  - difficult to load balance, possibly unneeded sync.
  - therefore: impractical
Parallel Tasks (1)

- Introduced in OpenMP 3.0 (May 2008)
- Allows to parallelize irregular problems like
  - unbounded loops (e.g. while loops)
  - recursive algorithms
- Unstructured parallelism
- Dynamically generated units of work
- Task can be executed by any thread in the team, in parallel with others
- Execution can be immediate or deferred until later
- Execution might be suspended and continued later by same or different thread
Parallel Tasks (2)

- Parallel threads enter a pool
- Tasks are executed as soon as threads become available
- Order is unpredictable

Source: R. van der Pas, Overview of OpenMP 3.0
iwomp.zih.tu-dresden.de/downloads/2.Overview’OpenMP.pdf
Example: Pointer chasing in C notation

```c
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead;
        while (p) {
            /* create a task for each element of the list */
            #pragma omp task
            process (p); /* process the list element p */
            p = next(p);
        }
    }
}
```
Parallel Tasks (4)

- Single construct ensures that only one thread traverses the list.
- Single thread encounters *task* directive and invokes the independent tasks.
- “Task” construct gives more freedom for scheduling, can replace loops with if statements that are not well load-balanced.
- Parallel tasks can be nested within parallel loops or sections.
Parallel DO loops ("for" loops in C/C++) are often the most important parallel construct.

The iterations of a loop are shared across the team (threads).

A parallel DO construct can have different clauses like REDUCTION.

```
sum = 0.0
!$OMP PARALLEL DO REDUCTION(+,sum)
DO i = 1, n
    sum = sum + a(i)
END DO
!$OMP END PARALLEL DO
```
Parallel Loops and Load Balancing

Example of a parallel loop with *dynamic* load-balancing:

```c
 !$OMP PARALLEL DO PRIVATE(i,j), SHARED(X,N),
 !$OMP& SCHEDULE (DYNAMIC,chunk)
 DO i = 1, n
   DO j = 1, i
     x(i) = x(i) + j
   END DO
 END DO
 !$OMP END PARALLEL DO
```
Parallel Loops and Load Balancing

- Iterations are divided into pieces of size *chunk*.
- When a thread finishes a piece, it dynamically obtains the next set of iterations.
- **DYNAMIC** scheduling improves the load balancing, default: **STATIC**.

**Tradeoff: Load Balancing and Overhead**
- The larger the chunk, the lower the overhead.
- The smaller the size (granularity), the better the dynamically scheduled load balancing.
Loops can be collapsed via the clause COLLAPSE

```fortran
!$OMP PARALLEL
!$DO COLLAPSE(2)
DO k = 1, p
  DO j = 1, m
    DO i = 1, n
      x(i,j,k) = i*j + k
    END DO
  END DO
END DO
!$OMP END DO
!$OMP END PARALLEL
```
Loop Collapsing (2)

- Iteration space from the two loops is collapsed into a single one

- Good if
  - loops k and j do not depend on each other (no recursions)
  - execution order can be interchanged
  - loop limits p and m are small, #processors is large

- Rules:
  - perfectly nested loops (j loop immediately follows k loop)
  - rectangular iteration space (m independent of p)
Quiz: Is there something wrong?

**Assume:** 4 parallel shared memory threads, all arrays and variables are initialized.

```fortran
! start the parallel region
!$OMP PARALLEL PRIVATE(pid), SHARED(a,b,n)
! get the thread number (0..3)
pid = OMP_GET_THREAD_NUM()
! parallel loop
!$OMP DO PRIVATE(i)
DO i = 1, n
   A(pid) = A(pid) + B(i) ! compute
END DO
!$OMP END DO
! end the parallel region
!$OMP END PARALLEL
```
False Sharing Example

Suppose you have $P$ shared memory processors, with $\text{pid} = 0 \ldots P-1$

Each processor runs the Fortran code:

```fortran
DO i = 1, n
    A(pid) = A(pid) + B(i)
END DO
```

No read nor write (load and store) conflicts, since no two processors read or write same element, but:

Performance is horrible!
False Sharing Example

Reason:

- Several consecutive elements of A are stored in same cache line.
- In each iteration, each processor gets an exclusive copy of entire cache line to write to, all other processors must wait.
- B read-only, so sharing not a problem.

⇒ Can be avoided by declaring $A(c,0:P-1)$, where $c$ elements equal 1 cache line, and using $A(1,pid)$.

**False sharing** is usually obvious once pointed out, but very easy to write in and overlook. Avoid!
False Sharing Example

1D: \( A(0:P-1) \)

2D: \( A(c,0:P-1) \)

---

same cache line: cache conflicts

different cache lines (not shared)
Race Conditions

In a shared memory system, one common cause of errors is when a processor reads a value from a memory location that has not yet been updated.

- This is a race condition, where correctness depends on which processor performed its action first.
- Often hard to debug because the debugger often runs the program in a serialized, deterministic ordering.
- To insure that “readers” do not get ahead of “writers”, process synchronization is needed.
- DM systems: messages are often used to synchronize, with readers blocking until the message arrives.
- Shared memory systems: barriers, software semaphores, locks or other schemes are used.
Race Condition Example

Two PARALLEL SECTIONS:

!$OMP PARALLEL SECTIONS

!$OMP SECTION
A = B + C

!$OMP SECTION
B = A + C

!$OMP END PARALLEL SECTIONS

- Unpredictable results since the execution order matters.
- Program will not fail: Wrong answers without a warning signal!
Critical / Ordered Region

- All threads execute the code, but only one at a time
- Useful to avoid a race condition, or to perform I/O (I/O still has random order in case of CRITICAL)
- May introduce serialization: expensive

Stout and Jablonowski – p. 146/268
OpenMP: Traps

OpenMP is a great way of writing fast executing code and your gateway to special painful errors.

- OpenMP threads communicate by sharing variables.
- Variable Scoping: Most difficult part of shared memory parallelization
  - Which variables are *shared*
  - Which variables are *private*
- If using libraries: Use the *threadsafe* library versions.
- Avoid sequential I/O (especially when using a single file) in a parallel region: Unpredictable order.
OpenMP: Traps

Common problems are:

**False sharing:** Two or more processors access different variables that are located in the same cache line. At least one of the accesses is a “write” which invalidates the entire cache line.

**Race condition:** The program’s result changes when threads are scheduled differently.

**Deadlock:** Threads lock up waiting for a locked resource that will never become available.
Question: How would you distribute the work in a climate model?
Part II: ADVANCED TOPICS

In this part we examine hybrid and pipelined computing models and consider tools to help develop efficient parallel programs. We also discuss more advanced aspects of load balancing and the parallelization process.

We conclude with some comments about using parallel systems, and a review.
Many of today’s most powerful computers employ both shared memory (SM) and distributed memory (DM) architectures.

These machines are so-called hybrid computers.

Hybrid architectures are likely to prevail and become even more complicated: e.g. heterogeneous machines with accelerators like Graphics Processing Units (GPUs).

The corresponding hybrid programming model is a combination of shared and distributed memory programming (e.g. a selected combination of OpenMP, MPI, CUDA, OpenCL or OpenACC directives).
Memory Systems: Distributed Memory

- All memory is associated with processors.
- To retrieve information from another processor’s memory a message must be sent over the network.

Advantages:
- Memory is scalable with number of processors
- Each processor has rapid access to its own memory without interference or cache coherency problems
- Cost effective: can use commodity parts

Disadvantages:
- Programmer is responsible for many of the details of the communication
- May be difficult to map the data structure
- Non-uniform memory access (NUMA)
Memory Systems: Shared Memory

- Global memory space, accessible by all processors
- Memory space may be all real or may be virtual
- Consistency maintained by hardware, software or user

Advantages:
- Global address space is user-friendly, algorithm may use global data structures efficiently
- Data sharing between tasks is fast

Disadvantages:
- May be lack of scalability between memory and CPUs. Adding CPUs increases traffic on shared memory - CPU path
- User is responsible for correct synchronization
- Consistency can be subtle, errors difficult to locate
Hybrid Memory Architecture

The shared memory component is usually a cache coherent (CC) SMP node with either uniform (CC-UMA) or non-uniform memory access (CC-NUMA).

CC: If one processor updates a variable in shared memory, all the other processors on the SMP node know about the update.

The distributed memory component is usually a cluster of multiple SMP nodes.

SMP nodes can only access their own memory, not the memory on other SMPs.

Network communication is required to move data from one SMP node to another.
Multi-core (e.g. quad-core) chips standard, even in laptops

Typical: Several multi-core chips are used for the SMP node.
Multi-Cores and Many-Cores

General trend in processor development: **multi-core** to **many-core** with tens or even hundreds of cores

- **Advantages**
  - Cost advantage.
  - Proximity of multiple CPU cores on the same die, signal travels less, high core-to-core clock rate.

- **Disadvantages:**
  - More difficult to manage thermally than lower-density single-chip design.
  - Needs software (e.g. OS, commercial) support.
  - Multi-cores share system bus and memory bandwidth: limits performance gain. E.g. if single-core is bandwidth-limited, the dual core is only 30%-70% more efficient.
Dual/Triple Level Parallelism

Often: Applications have several natural levels of parallelism. Exploit shared memory parallelism by using OpenMP on an SMP node. Utilize accelerators if present.

- MPI performance degrades when
  - domains become too small
  - message latency dominates computation
  - parallelism is exhausted

- OpenMP
  - typically has lower latency
  - can maintain speedup at finer granularity

Drawback:

- Programmer must know MPI, OpenMP, CUDA, ...
- Code might be harder to debug, analyze and maintain
Hybrid Programming Model

- Combination of distributed and shared memory programming models
- **Most important: MPI and OpenMP**
  - Each MPI process is assigned to different SMP node
  - Explicit message passing between the nodes
  - Shared memory parallelization within an SMP node
  - Each MPI process is therefore a multithreaded OpenMP process
  - Accelerators: attached to SMP node, activated via directives, CUDA or OpenCL
  - Hybrid programming can give better scalability than pure MPI or OpenMP
- Requires thread-safe MPI libraries and compilers!

Stout and Jablonowski – p. 158/268
Hybrid Programming Strategy

- Recommended:
  - Limit MPI communication to serial OpenMP part (outside a parallel region)
  - Let the master thread (serial OpenMP part) communicate via MPI messages.

- Be careful: Creation of OpenMP threads causes overhead, minimize

- Example on the next page:
  - Scientific codes often have a \textit{big} outer loop, e.g. over time
  - Inside the big loop: more loops over e.g. space
  - Target the outermost loop that can be parallelized (time loop is serial)
First Implementation (in C notation)

```c
for (...) { /* time loop, serial */
    // Initialization
    { ... }

    // Computation
    #pragma omp parallel for /* create threads */
    { for (...) 
        { ... } }
    /* threads are destroyed*/

    // Communication
    MPI_Recv (...); MPI_Send (...);
}
```

- Not the most efficient code, OpenMP threads are created and destroyed many times, causes overhead
- Improve: create OpenMP threads only once
Improved Implementation

```c
#pragma omp parallel private (...) /* create threads */
for (...) { /* time loop, all threads count the time */
    #pragma omp single /* one thread initializes */
    // Initialization
    { ... }

    // Computation
    #pragma omp for /* all threads execute loop */
    { for (...)
        { ... } }
        /* implied barrier */

    #pragma omp master /* thread 0 handles MPI */
    // Communication
    { MPI_Recv (...); MPI_Send (...); }

    #pragma omp barrier /* explicit barrier */
}
```
Performance: Hybrid MPI/OpenMP

Is it worth it? The answer is **maybe**.

Example: Matrix multiplication on 8 nodes, each node has 8 processors (64 processors total)

Performance gains depend on cache utilization

Source: Ashay Rane, Dan Stanzione, 10th LCI International Conference on High-Performance Clustered Computing, 2009
Example: Grid Partitioning

- MPI across colored patches, OpenMP within patch
- Left: fragmented small MPI blocks, Right: big blocks

Which one is more efficient? Most likely the fragmented decomposition due to cache and load-balancing aspects
Real code is long, complex. How do we engineer the parallelization process?

Most of the discussion is in terms of converting from serial to parallel, but ideas apply when building parallel from scratch.

Usually there is a (perhaps vague) performance goal, not, per se, a parallelization goal.
Parallelization Process

1. Set Goals
2. Analyze, profile
3. Prioritize changes, develop test cases
4. Incrementally change
   - Correct?
     - Yes
       - Ready for use (until goals change)
     - No
       - Performance acceptable?
         - Yes
           - Ready for use (until goals change)
         - No
           - Go back to prior steps

Stout and Jablonowski – p. 165/268
Overview of Approach

*Incremental approach:* tackle a bit of the problem at a time so that one can recover from mistakes and poor attempts.

- **Verify:** Develop test cases and constantly check results.
- **Profile:** Determine where time being spent. May be coupled with modeling of code to determine where effort will yield most reward.
- **Check-point/restart:** Aids testing and debugging since some problems only occur late in the program execution.
Incremental Process: Shared Memory

Incremental parallelization easiest on shared memory. Portions not parallelized will slow the program but will at least be correct.

More difficult than shared memory, but some similar ideas.
Parallelization Steps — DM

First need to develop maps of major data structures and where used.

- Initially, all processors have the complete standard serial data structures (global data structures).
- As parallelize code and data structures (local data structures), develop serial $\rightarrow$ parallel & parallel $\rightarrow$ serial conversion routines (scaffolding).
- Verify correctness on test cases by showing serial $\rightarrow$ parallel $\rightarrow$ serial = serial for global data structures.
We’ll continue the discussion of load-balancing, looking at some more complicated problems.
Static Load Imbalance — Correlation

Suppose have digital image, need to determine types of vegetation on the island. Easy load-balance:
However ...

If pixel is water can quickly dismiss it, otherwise need to carefully analyze pixel and neighbors.

Large regions will be of one type or the other. Thus some processors will take much longer than others.
However ...

If pixel is water can quickly dismiss it, otherwise need to carefully analyze pixel and neighbors.

Large regions will be of one type or the other. Thus some processors will take much longer than others.

Drat! We know the weights, representing computation, but we don’t know where the easy or hard pixels are until we’ve started processing the image.
Scattered Decomposition

Suppose there are $P$ processors. Partition image into grids of size $P$ and assign each processor a cell in each grid.
How Much Scattering?

More pieces ⇒

⇓ load imbalance, i.e., ⇓ calculation time
↑ overhead and/or communication time

Deciding a good tradeoff may require some timing measurements.
How Much Scattering?

More pieces ⇒

↓ load imbalance, i.e., ↓ calculation time

↑ overhead and/or communication time

Deciding a good tradeoff may require some timing measurements.

However, if nearby objects have uncorrelated computational requirements then this method is no better than standard decomposition, and adds overhead.
Overdecomposition

Scattered decomposition and its close relatives *striping* and *round robin allocation* are examples of a general principle:

*Overdecomposition*: break task into more pieces than processors, assign many pieces to each processor.

Overdecomposition underlies several load-balancing and parallel computing paradigms.

However, there can be difficulties when synchronization is involved.
The (Teaching) Value of Coins

Task times are random variables, where the time is generated by flipping a coin until a head appears.

- Your task times: ____________________________

- Class task times: ____________________________

- Your total: ____________

- Class total: ____________

- Slowest person’s total: ____________
Synchronization and Imbalance

- Suppose have \( p \) processors and \( n \geq p \) tasks.
- Tasks take time \( i \) with probability \( 2^{-i} \), and no way to tell in advance how long a task will take.
- If each processor does 1 task and then waits for all processors to complete before going on to the next, efficiency is low, decreasing as \( \log p \).
- No matter what the distribution of task times:
  - To improve efficiency, each processor needs to complete several tasks before synchronizing.
  - Unfortunately, \# of tasks/proc needed grows with \( n \). Weak scaling won’t give linear speedup.
Dynamic Data-Driven

For many data dependent problems dynamic versions also occur, such as

- For PDEs an adaptive grid can be used instead of a fixed grid, allowing one to focus computations on regions of interest.
- A simulation may track objects through a region.
- Computational requirements of objects may change over time.

In such situations, some processors may become overloaded.
Must balance load and need to take locality of communication into account. Some options:

- Locally adjust partitioning, such as moving small region on boundary of overloaded processor to processor containing the neighboring region.
- Use a parallel rebalancing algorithm that takes current location into account (not standard).
- Rerun the static load-balancing algorithm and redistribute work (ignores locality, but easier)

**Warning:** Need more complex data structures which can move pieces and keep track of neighbors, etc. These are difficult to program and debug.
Adaptive blocks, useful for adaptive mesh refinement (AMR), dynamic geometric modeling. Grids broken into blocks of fixed extents, when needed blocks refined into children with same extents. [Stout 1997, MacNeice et al. 2000]
Adaptive Block Properties

Whenever refine/coarsen occurs, must adjust pointers on all neighbors, no matter what processor they are on.

Using blocks, instead of cells, reduces the number of changes.

Same work per block, good work/communication ratio, so often just balancing blocks per processor suffices. If communication excessive use space-filling curve.

In either case, rebalancing requires only simple collective communication operations to decide where blocks go.
Load-balancing Strategies

Example: Tracer transport problems with adaptive mesh refinement (AMR) techniques

- Simple load-balancing algorithm:
  - Equal workload regardless of the location of the data

- Advanced load-balancing algorithms:
  - Load-balancing with METIS
  - Load-balancing with a Space Filling Curve (SFC)

⇒ In the examples:
- Each color represents a processor.
- The amount of work in each box is the same.
Simple Load-balancing Strategy
Simple Load-balancing Strategy cont.

Data distribution at model day 3:
Simple Load-balancing Strategy cont.

Data distribution at model day 12:
Dynamic Load-balancing

Metis

Space Filling Curve

Courtesy of Dr. Joern Behrens, University of Hamburg, KlimaCampus, Germany
Comparison of Strategies

Relative behavior similar to static load-balancing behavior. Very important that rebalance operations have low overhead since they will be done often.

- Easiest strategy — just balance work/processor
  - might be sufficient if application is dominated by computation, but not if communication important

- Load-balancing with METIS or ParMETIS
  - good load-balancing, decent comm. reduction, applicable to many problems

- Load-balancing with Space Filling Curves
  - for geometric problems usually the best choice
Dynamically Generated Work

- Sometimes work created on the fly with little advance knowledge of tasks.
- E.g., branch-and-bound generates dynamic partial solution trees where subproblem communication consists of maintaining a current best solution and seeing if subproblem already solved.
- In such situations can maintain a queue of tasks (objects, subproblems) and assign to processors as they finish previous tasks (e.g., overdecomposition).
Example: Work Preassigned

Each processor is assigned 4 tasks.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Task Label/Time</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a/5 b/1 c/1 d/4</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>e/1 f/4 g/2 h/1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>i/2 j/1 k/5 l/1</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>m/1 n/3 o/1 p/1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>q/1 r/1 s/2 t/2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>u/3 v/4 w/2 x/3</td>
<td>12</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

Time required: 12.
Manager/Worker (Master/Slave) (them/you)

Manager

Task Queue

assign tasks

worker

worker

worker

worker

Task done

request another
Work Assigned via Queue

Assign tasks a, b, c, ... to processors as the processor becomes available:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time / task assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a a a a a a r v v v v</td>
</tr>
<tr>
<td>2</td>
<td>b g g k k k k k k k</td>
</tr>
<tr>
<td>3</td>
<td>c h j l n n n n w w</td>
</tr>
<tr>
<td>4</td>
<td>d d d d o s s x x x x</td>
</tr>
<tr>
<td>5</td>
<td>e i i i m p t t t</td>
</tr>
<tr>
<td>6</td>
<td>f f f f f q u u u u</td>
</tr>
</tbody>
</table>

Time: **10.** Adaptive allocation can improve performance.
Work Assigned via Ordered Queue

Sort in decreasing order, assign to processors as they become available.  

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time / task assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>1</td>
<td>a a a a a s s e o</td>
</tr>
<tr>
<td>2</td>
<td>k k k k k t t h p</td>
</tr>
<tr>
<td>3</td>
<td>d d d d x x x j q</td>
</tr>
<tr>
<td>4</td>
<td>f f f f g g w w r</td>
</tr>
<tr>
<td>5</td>
<td>v v v v v i i b l</td>
</tr>
<tr>
<td>6</td>
<td>n n n u u u c m</td>
</tr>
</tbody>
</table>

Time: 9.  The more you know, the better you can do. Unfortunately, rarely have this information.
Queueing Costs

- Single-queue multiple-servers (manager/workers) most efficient queue structure (e.g., airline check-in lines).

- However, queuing imposes communication overhead, yet another tradeoff: cost of moving task versus cost of solving it where it is generated.
Queueing Costs

- Single-queue multiple-servers (manager/workers) most efficient queue structure (e.g., airline check-in lines).

- However, queuing imposes communication overhead, yet another tradeoff: cost of moving task versus cost of solving it where it is generated.

Parallel computing has too many “however”s!
Queueing Costs

- Single-queue multiple-servers (manager/workers) most efficient queue structure (e.g., airline check-in lines).

- However, queuing imposes communication overhead, yet another tradeoff: cost of moving task versus cost of solving it where it is generated.

Parallel computing has too many “however”s!

However, if it was too easy, you wouldn’t need this tutorial
Queueing Bottleneck

Sometimes the manager is a bottleneck. In OpenMP loops, assigning iterations to cores is a manager-worker problem.

- “Chunk” tasks to reduce overhead.
  - OpenMP loops: SCHEDULE(DYNAMIC,chunk)
  - Might use large chunks initially, then decrease them near the end to fine-tune load balance.
  - OpenMP loops: SCHEDULE(GUIDED,chunk)

- Use distributed queues. Many variations:
  - multiple manager/worker subteams, with some communication between managers
  - every worker is also a manager, keeping some tasks and sending extras to others.
Load-balancing is *critical* for high performance.

Depending on the application, can range from trivial to nearly impossible. A wide range of approaches are needed, and new ones are constantly being developed.
Load-balancing is critical for high performance.

Depending on the application, can range from trivial to nearly impossible. A wide range of approaches are needed, and new ones are constantly being developed.

Try simple approaches first.
Accelerators gain popularity: www.top500.org
Graphics Processors Become Versatile

http://www.ornl.gov/ornl/news
**Accelerator: Pipelining Principle**

**Principle:** Split an operation into independent parts & execute them concurrently in specialized pipelines / accelerators

- **Add pipeline**

  ```
  DO I = 1, 1000  
  C(I) = A(I) + B(I)  
  ENDDO
  ```

- **Accelerators like Graphics Processing Units (GPUs) are specialized for such SIMD-like operations.**

- **Extremely fast.**

Source: [www.forwardlook.net/images/Photo09.jpg](http://www.forwardlook.net/images/Photo09.jpg)
Multifunctional pipelines are also used in most scalar processors ⇒ speed up the code

**Example:**

**IBM Power7 CPU:** Floating point units (FPU) which can issue a combined multiply/add

\[ a = b \times c + c \]

- Multi-functional hardware unit: 2 floating point operations per clock cycle
- Processors also have data prefetch capabilities ("load pipeline")
Graphics Processing Units (GPUs)

- GPUs are cheap: produced in the millions.
- GPU contains a graphics pipeline, is an accelerator.
- GPUs have a parallel many-core architecture, each core capable of running thousands of threads.
- SIMD fine-grain parallelism
- Traditionally: GPU dedicated graphics rendering device for a personal computer or game console, often integrated into motherboard, have become General Purpose GPUs (GPGPUs).
- Highly parallel structure makes them more effective than general-purpose CPUs for some algorithms.
Each Arithmetic Logic Unit (ALU) has a control unit, private cache and shared DRAM memory.

Pipelines in GPUs can typically utilize three levels of parallelism:
- (1) Data (Data/Loop), (2) Task, (3) Pipeline

GPU might have reduced or specialized instruction set.
Heterogeneous Architectures

- Trend: **Highly diverse** computing platforms can include multi-cores, SMP nodes, and accelerators as co-processors.

- Many of the most powerful systems are heterogeneous:
  - #1 on TOP500 6/2013: “Tianhe-2” (Milky Way 2) with Intel’s Xeon Phi accelerator
  - #2 on TOP500 6/2013: “Titan” with NVIDIA’s Kepler GPU

- Caveat: Very difficult to use the heterogeneous hardware effectively.
Examples of Accelerators

- NVIDIA’s Kepler GPU (K20X)
  - 1.31 TFlops double precision performance, 3.95 Tflops single prec.
  - 2688 cores
  - 250 GB/s memory bandwidth, 6 GB RAM

- Intel’s Xeon Phi 7120X (Many Integrated Cores (MIC) architecture)
  - 1.2 TFlops double precision performance
  - 61 cores (244 threads)
  - 352 GB/s memory bandwidth, 16 GB RAM
Currently: Host CPU handles all data transfers (copies) to and from the accelerator (can be a bottleneck)

Future: Accelerators might have direct access to CPU memory, might be able to communicate without host
How to Program an Accelerator

- **Low-level programming model:** Major code re-write
  - Proprietary programming languages or extensions like NVIDIA CUDA C/C++, Portland Group (PGI) CUDA Fortran (not portable, only NVIDIA GPUs)
  - Portable: OpenCL (Open Computing Language), [http://www.khronos.org/opencl/](http://www.khronos.org/opencl/)

- **High-level programming model:** Accelerator compiler directives
  - OpenMP 4.0 support for accelerators [www.openmp.org/mp-documents/OpenMP4.0.0.pdf](http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
  - Single code base: Compile same program on multi-core CPU or CPU/GPU systems, portable

- **Big question:** Is it worth your time (yet) developing accelerator software?

Stout and Jablonowski – p. 205/268
OpenMP Accelerator Principles

- Host-centric model: host device “offloads” code regions and data to accelerators for execution, specified using the `target` construct.

- A `device` (any logical execution engine)
  - has independent shared memory
  - has threads that behave almost the same as threads on the host device

- `map` clause: defines how variables are handled by the device, including allocation, initialization and assignment to the host variables at the end of a `target [data]` region
OpenMP Accelerator Example (1)

- Works well on *Intel Xeon Phi*

```c
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

- Environment Variable `OMP_DEFAULT_DEVICE=<int>`: sets device number for target constructs
- Here: Device 0 is the accelerator
- Array B will be copied from host to the accelerator and back to host
OpenMP Accelerator Example (2)

- Works well on *NVIDIA GPUs*

```c
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(nb) num_threads(nt)
#pragma omp distribute
for (i=0; i<N; i+= nb)
    #pragma omp parallel for
    for (b = i; b < i+nb; b++)
        B[b] += sin(B[b]);
```

- Number of parallel teams: \(nb\)
- Number of parallel threads in each team is \(nt\)
- *distribute*: Iteration of loop will be executed by the thread teams
How Much Speedup Can Be Expected?

- Be realistic
- Some isolated computing kernels can show significant speedups on GPUs
- Speedup factors in the range of 10X-100X have been reported on these kernels
- In reality:
  - only some fraction of a real application can utilize the GPU effectively
  - host-to-GPU memory copies are expensive (slow)
  - often: multi-core CPUs utilization can be further optimized
  - more realistic: GPU application shows speedups around 2X-4X
GPU Memory Bandwidth

- Memory is a critical resource in GPUs
- GPUs have limited off-chip memory access bandwidth compared to peak compute throughput
- Example: NVIDIA’s Kepler GPU (K20X)
  - 1.2 TFlop/s peak double precision (DP) throughput
  - 250 GB/s peak off-chip memory access bandwidth
  - Corresponds to $250 \text{ GB/s} / 8 = 31 \text{ G DP (8 Bytes)}$ operands per second
- To achieve peak throughput, a program must perform $1200 / 31 \approx 38$ DP floating point arithmetic operations for each operand value fetched from off-chip memory
If operands are only used once, performance is limited by the memory bandwidth.

Then we only get a peak of $\frac{1200 \text{ GFlops}}{38} = 31 \text{ GFlops (DP)}$ on Kepler GPU.

Ideally, we want... In reality, we might get...
Accelerator Trends

- Intel’s MIC co-processor technology: supports standard programming model OpenMP & MPI, also supported by PGI OpenACC directives

- GPU high-level programming models evolve rapidly:
  - OpenACC directives (e.g. implemented in PGI Accelerator compilers)
  - Newest OpenMP 4.0 (July 2013) standard now includes specifications for GPU directives
  - Future (likely): OpenACC will be adopted by / merged with OpenMP to create a single standard
  - Currently: very limited compiler support for OpenMP 4.0 accelerator directives, will change

- Programming model is a moving target: Should you wait?
"Is this a good time to tell you I don’t know what ‘big data’ means?"

Credit: John Klosser
Massive data collections important for Walmart, Square Kilometer Array, Google, bioinformatics, NSA . . .

Databases and “Big Data” important commercial application of parallel computers.

Disk access and bandwidth, not flops, dominate performance.

Graph500 uses a benchmark more useful for complex data analytics. [http://www.graph500.org](http://www.graph500.org)
“Shared Some” or “Shared Disk”

Shared disk system often managed separately, may have multiple systems attached.

For some unstructured applications, massive shared memory gives big improvement.
Organizing Data

- Classic relational databases based on indexed tables
  - Organizing data to match access patterns is critical
  - This has been studied and optimized for decades.

- “Big data” typically not well-structured databases.
  - May have billions of small files.
  - Access and indexing may change over time.
  - NoSQL vs. SQL (Structured Query Language)

- Parallel I/O also important for numeric calculations
  - MPI provides this
  - pnetCDF and HDF5 portable formatting schemes
    (parallel network Common Data Form and Hierarchical Data Format)
Map-Reduce

- A new(?) form of data mining often used on big data
- Variations used by Google, Yahoo, IBM, etc.
- Implementations have significant emphasis on locality, scalability and fault tolerance

- Open source Hadoop:  

- Hadoop available on NSF’s Extreme Science and Engineering Discovery Environment (XSEDE) FutureGrid  

- Tutorial:  
Map-Reduce Example

Given records (student name, SAT score, web link):
For every university find # times a student with SAT > 1550 viewed one of the university’s web pages

**Map:** for every student with SAT > 1550, if link points to a university then generate new record
(university, 1)

Operation embarrassingly parallel, I/O bound

**Reduce:** combine records by school and sum the counts.
Requires communication, but far fewer records.
Reduction allows flexibility in ordering calculations.
Basic Map-Reduce Process

This is logical view, implementation typically complex
Further Functionality

- Mapping and reduction much more general. E.g., for each school find names of all students that viewed more than one of the school’s web pages.

- A single record may generate many tuples, e.g., document & words it contains

- Hadoop provides dynamic load balancing, extensive reporting available on progress and efficiency, etc.
Developing large-scale scientific or commercial applications that make optimum use of the computational resources is a challenge.

Resources can easily be underutilized or used inefficiently.

The factors that determine the program’s performance are often hidden from the developer.

Performance analysis tools are essential to optimizing the serial or parallel application.

Typically measured in "Floating point operations per second" like Mflops, Gflops, Tflops or Pflops.
Application-System Interplay

System factors:
- Chip architecture (e.g. # floating point units per CPU)
- Presence of co-processors (accelerators)
- Memory hierarchy (register - cache - main memory - disk)
- I/O configuration
- Parallel file system (supporting parallel I/O)
- Compiler
- Operating system
- Connecting network between processors
Application-System Interplay

Application factors:
- Programming language (C/C++, Fortran, CUDA, ...)
- Algorithms and implementation
- Data structures
- Memory management
- Libraries (e.g. math libraries)
- Size and nature of data set
- Compiler optimization flags
- Use of I/O
- MPI / OpenMP / Accelerators directives
- Communication pattern
- Task granularity
- Load balancing
Performance Gains: Hardware

Projected Performance Development

Factor $\approx 10^5$ increase over the last 20 years

Source: Top500 list, http://www.top500.org/
Performance Gains: Software

- Gains expected from better algorithms (here numerical linear algebra), load-balancing, parallel I/O, etc.

Source: A science-based case for large-scale simulation, DoE Office of Science, 2003 a.k.a. SCaLeS report Vol.1
Optimize Serial Performance First

Eliminating serial performance problems is critical to attaining parallel performance goals.

*Doubling serial performance is far more useful than doubling the number of processors*

Examples:

- Access data continuously in memory (optimize the cache use)
- Reuse data in cache, utilize CPU pipelines
- Avoid if-statements and procedure calls in loops
- Minimize dynamic memory allocation (slow)
Serial Efficiency Example: Cache Misses

Many programs have excessive loads and stores, causing cache misses which slow the program. Can often be reduced by rearranging the code and/or data structure.

For example, in Fortran

```fortran
  do i=1,n
    do j=1, n
    enddo
  enddo

  do j=1, n
    do i=1,n
    enddo
  enddo
```

For large arrays, which is faster, and why?
For a well-structured program it should be possible for the compiler to generate good code — optimizing cache utilization, reducing instruction counts, etc. However, extensive optimization is \textit{not the default}. Thus

Turn on appropriate compiler optimization options.

Usually “O” option important, but often others needed as well. These affect data placement as well as code generation.
Performance Gains: Utilize the Compiler

For a well-structured program it should be possible for the compiler to generate good code — optimizing cache utilization, reducing instruction counts, etc. However, extensive optimization is *not the default*. Thus

Turn on appropriate compiler optimization options.

Usually “O” option important, but often others needed as well. These affect data placement as well as code generation.

*May need a guru to get best combination of options for your program+machine combination.*
Parallel Performance Analysis

- Reveals not only typical *bottleneck situations* but also determine the *hotspots*

- Key question: How efficient is the parallel code?

- Important to consider: Time spent
  - communicating to other processors
  - waiting for a message to be received
  - wasted waiting for other processors

- When selecting a performance tool consider:
  - How accurate is the technique?
  - Is the tool simple to use?
  - How intrusive is the tool?
Parallel and Serial Performance Analysis

**Goal:** reduce the program’s wallclock execution time

**Practical, iterative approach:**

- measure the code with a hardware performance monitor and profiler
- analyze hotspots
- optimize and parallelize hotspots and eliminate bottlenecks
- evaluate performance results and improve optimization / parallelization

**Analysis techniques**

- Timing (e.g. MPI_Wtime)
- Counting (hardware counter)
- Profiling
- Tracing
Hardware Performance Monitors (HPM)

Hardware counters gather performance-relevant events of the microprocessor **without** affecting the performance of the analyzed program. Two classes:

**Processor monitor:**
- non-intrusive counts
- consists of a group of special purpose register
- registers keep track of events during runtime: floating point efficiency, cache misses, branch miss prediction, memory access patterns
- **measures Flops fairly accurately**

**System level monitor (bus and network monitor):**
- bus monitor: memory traffic, cache coherency
- network monitor records network traffic
PAPI: The Portable Performance API

- mature public-domain Hardware Performance Monitor
- version Papi 5.2.0 released in 8/2013
- vendor independent hardware counter tool
- supports most current processors including accelerators
- user needs to instrument code ⇒ PAPI functions
- Fortran and C/C++ user interfaces
- easy-to-use and powerful high level API
- also used by many tracing tools like Vampir, Open|SpeedShop

Home page:

http://icl.cs.utk.edu/papi/
Profiling of Parallel Programs

- Profilers identify **hotspots**

- Simplest tool: UNIX profiler **gprof** (public domain)
  - interrupts program execution at constant time intervals
  - counts the interruption
  - the more interruptions the more time spent in this part of the code
  - sum of all processors is displayed

- Commercial tool: **Allinea MAP**
  [http://www.allinea.com](http://www.allinea.com)

- Public domain Tuning and Analysis Tool **TAU**:
  [http://www.cs.uoregon.edu/research/tau](http://www.cs.uoregon.edu/research/tau)
Tracing Tools for Parallel Codes

- Collect trace data at run time, display post-mortem
- Assess performance, bottlenecks and load-balancing problems in MPI & OpenMP & Accelerator codes
- Intel’s trace visualization tool **Trace Analyzer & Collector** (only on Intel platforms)
- **Vampir** and **Vampirtrace** (platform independent)

Trace analyzer developed and supported by the Center for Information Services and High Performance Computing, Dresden, Germany

([http://www.vampir.eu](http://www.vampir.eu))

- Free evaluation keys for both available online.
Trace Analyzer & Collector / Vampir

**Trace Analyzer / Vampir** graphical user interface helps
- understand the application behavior
- evaluate load balancing
- show barriers, locks, synchronization, I/O
- analyze the performance of subroutines/code blocks
- learn about communication and performance
- identify communication hotspots

**Trace Collector / Vampirtrace**
- Libraries that trace MPI and application events, generate trace file (files can become big!)
- Convenient: Re-link your code and run it
- Provides API for more detailed analyses
Vampir Analysis – Customizable Displays

Source: http://www.vampir.eu/tutorial/manual
Vampir Analysis – Zoom-in Timeline

Reveals communication and synchronization patterns
Vampir Analysis – Process Summary

Accumulated time: work load, load imbalances
Vampir Analysis – Performance Data

Performance data (Gflops) via hardware monitor PAPI
Vampir Analysis – MPI Communication

Communication pattern and message sizes
Public Domain Tools

TAU – Tuning and Analysis Utilities (version 2.22.2)

- Developed at the University of Oregon, mature
- Free, portable, open-source profiling/tracing facility
  (http://www.cs.uoregon.edu/research/tau)
- Performance instrumentation, measurement and analysis toolkit for MPI, OpenMP and Accelerator codes
- Manual or automatic source code instrumentation
- Utilizes the scalable Open Trace Format (OTF): also viewable by Vampir

OpenSpeedShop, performance tool by Krell Institute
  (http://www.openspeedshop.org)
Performance Analysis: Strategy

- **Hardware counters/Vampir** provide information on Flops rates, do you need to optimize?
- Use **profilers** to identify hotspots
- Focus the analysis/optimization efforts on the **hotspots**
- Analyze **trace information**: gives detailed overview of the parallel performance, load-balance and reveals bottlenecks
  - two different modes: the *uninstrumented* or *instrumented* mode (requires source code changes)
    ⇒ Pitfall: can lead to huge trace files
  - Recommendation: instrument only hotspots for detailed view of the run time behavior
Debugging of Parallel Programs

- **Higher parallel complexity**: debugging more difficult.
- Traditional sequential debugging technique is cyclic approach where the program is repeatedly stopped at breakpoints and then continued or re-executed again.
- Conventional style of debugging sometimes difficult with parallel programs: they do not always show reproducible behavior, e.g. race condition.
- Always: turn on compiler debugging options like array-bound checks

Most powerful commercial debuggers:
- **Allinea DDT** ([http://www.allinea.com/products/ddt](http://www.allinea.com/products/ddt))
Characteristics of Totalview

- Very powerful and mature debugger
- Source-level, graphical debugger for C/C++, Fortran
- Multiprocess (MPI), multithread (OpenMP) and accelerator (OpenACC/CUDA) codes
- Designed for UNIX platforms
- Intuitive, easy-to-learn graphical interface
- Industry leader in MPI, OpenMP, coprocessor debugging
- Control functions to run, step, breakpoint, interrupt or restart a process or coordinated groups of processors
- Ability to control all parallel processes coherently

Good tutorial on TotalView with parallel debugging tips:
http://www.llnl.gov/computing/tutorials/totalview/
TotalView: The Process Window

- 5 panes
- zoom into code or variables
- visualize variables
- filter, sort or slice data
- set break-points
- scan parallel processes
- step by step execution
Graphical representation of the message queue state
⇒ Red = Unexpected, Blue = Receive, Green = Send

Stout and Jablonowski – p. 246/268
Boost the Performance: Practical Tips

- Turn on compiler optimization flags
- Search for better algorithms and data structures
- For scientific codes: use optimized math libraries
- Tune the program:
  - data locality and cache re-use within loops
  - avoid divisions, indirect addressing, IF statements, especially in loops
  - loop unrolling and function inlining (often compiler option), minimize/optimize I/O, ...
- Load-balance the code
- Avoid synchronization/barriers whenever possible
- Optimize partitioning to minimize communication
- Identify inhibitors to parallelism: data dependencies, I/O
Parallel Libraries & Software Tools

**Parallel math libraries**: Highly optimized, recommended

**NAG Parallel Libraries** (commercial, often installed):
- Mostly high speed linear algebra routines

**PETSc** (Portable, Extensible Toolkit for Scientific computation):
- Designed with MPI for partial differential equations

**Eclipse**: Parallel Tools Platform (PTP)
- open-source project: wide variety of parallel tools
- provides: highly integrated environment specifically designed for the development of parallel applications
- framework for: coding & analysis, launching & monitoring, debugging, and performance tuning

In addition to programming, there are many issues concerning the use of parallel systems.

For example, they is often a centralized resource that must be shared, like mainframes of olden days.
Batch Queuing

A flashback to the 60’s

- Large parallel systems use batch queuing, may allow small interactive jobs for debugging.

- If there are multiple queues, learn how they are structured and serviced — it’s you vs. them.

- If submit several jobs at once, it may be you vs. you. Might improve throughput by requesting fewer processors, and more time, per job (Amdahl’s Law). Often use smallest # proc. that can run job in RAM.
Access to Systems

Academics (USA): Can apply for free time via XSEDE. For modest requests the process is easy and quick.
https://www.xsede.org

Another source of large allocations: DOE INCITE
www.doeleadershipcomputing.org/incite-program/

Other agencies’ grants usually supply computing.

Academics outside US: Most nations have similar programs, e.g., Partnership for Advanced Computing in Europe (PRACE) http://www.prace-ri.eu

Businesses: Can purchase time from hardware vendors, sometimes from national or university centers.
WRAP UP

We’ll review some of the material learned, discuss some general problems with parallel computing, and point out some trends in the area.
Trends in Parallel Computing

It’s useful to have a sense of where it is going.
Trends in Parallel Computing

It’s useful to have a sense of where it is going.

Source: Top500 Presentation at ISC’13
Chip Trends

Sources: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Energy Constrains Options

Problem:
- Heat limits packing density, Power In = Heat Out
- RAM also uses power
- Largest systems use > $10M electricity/year

Solution (?):
- Power: typically power \( \approx \text{speed}^2 \), thus lower clock speed and use more cores to regain flops. Also reduce RAM/core.

- Tianhe 2: 3,120,000 cores, 0.33 GB RAM/core
- Exascale (\( \approx 2019 \)): \( 10^8 \) cores, ? GB/core

*Tradeoffs opposite programmer needs* — Amdahl’s law.
Flops/Watts

- flops/Watt improving, though not exponentially
- Green500: # 1 (Eurora): 3209 Mflops/W, # 500: 42.3

Source: Top500.org
AMTTI Data Collected From 21 Different LANL Platforms Show Remarkably Similar Trends

More Trends

- Tianhe-2 and Titan grab the headlines, but clusters are most important economically.
- Increasing use of commercial parallelized software.
- $/flop continues to plummet: for 1 Gflop
  1961: $\approx 1,100,000,000,000,000$.  Now: $\approx 0.20$ (GPU)
- Irregular data access increasingly important. Graph500, Sequoia is # 1.
  [http://www.graph500.org](http://www.graph500.org)
- Some parallel computing companies will fail.
Review: Degrees of Difficulty

Some problems much easier to parallelize than others. Classes of problems range from

**Embarrassingly Parallel:** Separate jobs with no interaction, easy to run on any system.

**Static:** Important load-balancing parameters, such as size, known in advance. Often run same configuration many times.

**Data-dependent Dynamic:** Often quite difficult to achieve efficient implementation.
All new codes should be developed for parallel since all new computers are parallel.

Standard languages (e.g., MPI, OpenMP) and tools reduce learning curve and preserve investment.

Parallelizing serial code, and developing new code
- Start with overview of data structures & time requirements, do profiling as needed.
- Prioritize sections to be parallelized, and adapt as you learn.
- Parallelize at the outermost loop possible.
- Proceed incrementally, constantly verify correctness.
Review: Efficiency

- Reduce communication costs:
  - maximize data locality
  - eliminate false sharing in shared memory systems
  - combine messages to reduce overhead and synchronization
  - send data (distributed memory) or write data (shared memory) early, receive or read late.

- Reduce load imbalance and synchronization.

- Utilize compiler optimizations, optimized routines, etc.
High Performance is Bleeding Edge

- Economics dictates commodity chips: multicore + GPGPU, or manycore CPGPU
- Need mixed mode programming, MPI + (OpenMP, CUDA, OpenCL, ...), complex to write and optimize
- Perhaps GAS languages (CAF, UPC) will help in some situations
- Need better compilers and run-time system to exploit parallelism that the programmer has exposed (e.g., much smarter OpenMP and GAS compilers)
- Need new approaches and better algorithms
If It Isn’t Working Well . . .

- Early approaches probably weren’t developed for parallelism

- See if there is a more parallelizable approach

Credits: eMercedesBenz.com, boldride.com, Library of Congress
Sometimes parallelizable approaches aren’t the most efficient ones for serial computers, but that is OK if you are going to use many processors.
Sometimes parallelizable approaches aren’t the most efficient ones for serial computers, but that is OK if you are going to use many processors.

Remember Amdahl’s Law:

Efficient massive parallelism is difficult.
Sometimes parallelizable approaches aren’t the most efficient ones for serial computers, but that is OK if you are going to use many processors.

Remember Amdahl’s Law:

Efficient massive parallelism is difficult.

Make sure the goals are realistic, and remember that your own time is valuable.
Selected web resources for parallel computing are (occasionally) maintained at

http://www.eecs.umich.edu/~qstout/parlinks.html
References


Graph500: see Rankings

Green500: see Rankings


Hilbert space-filling curve: see the routines available in Zoltan (listed below).


Metis and Parmetis: [http://glaros.dtc.umn.edu/gkhome/metis/metis/overview](http://glaros.dtc.umn.edu/gkhome/metis/metis/overview)
References continued

MPI: documentation at http://www.mpi-forum.org/
Free, portable versions at:
http://www.mpich.org
http://www.open-mpi.org/

OpenACC: http://www.openacc.org

OpenCL: http://www.khronos.org/opencl/

OpenMP: http://openmp.org/wp/.

Parallel computing, a slightly whimsical explanation
http://www.eecs.umich.edu/~qstout/parallel.html

Rankings of supercomputers

Top500: http://www.Top500.org, is the longest running and best known. It is based on achieved number of flops on the Linpack benchmark.

Green500: http://www.green500.org, uses rankings based on flops/watt, a metric which is increasingly important.

Graph500: http://www.graph500.org, uses performance on sparse graph problems, a benchmark more suitable for problems such as data analytics.
References continued

- Top500: see Rankings.
- XSEDE, Extreme Science and Engineering Discovery Environment, a free parallel computing systems available to academics in the US: [https://www.xsede.org](https://www.xsede.org)