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  Big Data & Data Intensive Computing
         Hybrid Computing
         Load Balancing II
         Accelerators
         Performance Improvement, Debugging, Tools
         Using Parallel Systems
         Wrapup
INTRODUCTION

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 for complex problems
- Exabytes of storage in data centers
- 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:
  assembly lines, ant colonies, wolf packs, tutorials, …
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.
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 + massive storage + visualization.

Pipelining (streaming)  Breaking a task into steps performed by different units, with inputs streaming through, much like an assembly line.
Pipeling, Detroit Style

Ford assembly line
Who Uses Supercomputers?

- Energy
- Automotive
- Finance
- Database
- Telecomm
- Weather & Climate
- Aerospace
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.
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.
- Note that in a crash elements may end up touching that weren’t touching initially.
- The *state* of an element is its location, velocity, and other information such as whether it is metal that is bending.
The Car of the Future

(human driver optional)
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!
Chopping up a Car

How are elements assigned to processors?

Typically element assignment determined by serial preprocessing, using domain decomposition approaches (load-balancing) described later.
Keeping the Pieces Connected

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.
Evaluating Parallel Programs

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.

Tools to help do this will be discussed this afternoon.
Definitions

For a given problem A, let

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

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

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 \( p = \sqrt{n} \).
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:

$$Speedup \leq 1/f$$
Unfortunately, he found that typically $f$ was 10 – 20%, i.e., in best possible parallelization, 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 $\text{ParTime}(S) = \text{SerTime} \cdot [f + (1-f)/S] = \text{SerTime} \cdot [2f - f^2]$

$\approx \text{SerTime} \cdot 2f = \text{SerTime} \cdot 2/S$
Maximum Possible Performance
Amdahl Was an Optimist

Parallelization usually adds communication.

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

Amdahl’s Law

New Work
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.
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., as $n \uparrow$, Amdahl’s “$f$” \downarrow
Scaling

- Can often exploit large parallel machines by increasing \( n \) as \( p \) increases

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

- Amdahl considered \textit{strong scaling}: \( n \) is fixed

- 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. Further, essentially nothing scales to arbitrarily many processors.

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

\[
\begin{cases}
\{ S \\ M \} & I & \{ S \\ M \} & D
\end{cases}
\]

**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 in the past.

MIMD: almost all parallel computers are of this form. *This is the primary focus of the tutorial.*

MISD: extremely rare

SIMD: data level parallelism, there used to be large SIMD/vector systems in the past. Today:
- GPUs have SIMT units (T="threads")
- Most commodity CPUs feature instructions for a form of vector processing on multiple data, e.g. Intel & AMD have Advanced Vector Extensions (AVX)
A Conceptual View of SIMD

Controller, with program

Instruction

Processors, with data
**SIMD from a Software Viewpoint**

*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.

*Example*: A climate model (Single Program) that consists of an atmospheric, ocean and ice component with independent data and instructions (Multiple Data) in each component.

Dominant form of parallel programming.
Shared vs. Distributed Memory

SHARED MEMORY

- Memory
- Network
- Processor + cache

DISTRIBUTED MEMORY

- Memory
- Network
- Processor + cache + memory
Shared Memory (SM)

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

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

Disadvantages:
- Shared memory - to - CPU path may be a bottleneck (is bandwidth of the network sufficient?)
- Often: Non-Uniform Memory Access (NUMA) ⇒ access time varies, depends on physical distance
- Programmer responsible for correct synchronization
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
SM/DM Hardware/Software

Problem: programmers prefer SM, accountants prefer DM. *Software and hardware models do not need to match*

DM software on SM hardware:
- Message Passing Interface (MPI, discussed later) is for DM programming, but always available on SM systems. MPI messages become fast memory copies.

SM software on DM hardware:
- MPI-3 includes Remote Memory Access (RMA) options
- *Partitioned global address space (PGAS)* languages simulate SM. Examples: Unified Parallel C (UPC), Co-Array Fortran (CAF) (part of Fortran 2008).
- However, PGAS efficiency is often problematic because they are trying to mask significant latency.
Communication Network

There are many networks, but for the user differences are 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.

[http://www.youtube.com/embed/ystkKXzt9Wk?rel=0](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
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.
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 (µsec), 8 Flops/cycle: 20800 Flops per µsec.
- 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 are *blocking*, i.e., if processor cannot proceed until the message is finished, then can reach *deadlock*, where no processor can proceed.

*Example*: 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.

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

- International standard
- MPI evolves: MPI 1.0 was first introduced in 1994, most current version is MPI 3.1 (from June 2015)
- 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

While MPI large (hundreds of 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.citutor.org/browse.php

Writing MPI-based parallel codes helps preserve your investment as systems change.
MPI Basics

- 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.

- However: often a system buffer is used that allows small messages to be non-blocking send-receive handshakes, but large messages will be blocking.

- MPI implementation (not the MPI standard) decides this

- Blocking communication can be unsafe and may lead to deadlocks.

Stout and Jablonowski – p. 63/250
MPI Communication: Two Possibilities

- Small messages make use of system-supplied buffer

  Process 0  
  User data  
  Local buffer  
  the network  

  Process 1  
  Local buffer  
  User data

- Large messages need user-supplied receive buffer

  Process 0  
  User data  
  the network  

  Process 1  
  User data
Sources/Avoidance of Deadlocks

Incorrect order for blocking sends and receives

- Often works for small messages, fails for big messages

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Send (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Recv (0)</td>
</tr>
</tbody>
</table>

- “Unsafe” because completion/deadlock depends on the availability & size of system buffer, copies are also slow.

Correct order for blocking sends and receives

- User in control of send-recv handshake, avoids deadlock

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Recv (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Send (0)</td>
</tr>
</tbody>
</table>

Stout and Jablonowski – p. 65/250
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 system can provide the overlap.
MPI Initialization

Near the beginning of the program, include

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

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.
MPI Example

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
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 (user-defined) data types also possible:
  - Vector: data separated by constant stride
  - Contiguous: vector with stride 1
  - Struct: general mixed types (e.g. for C struct)
  - Indexed: Array of indices
Improving Performance

In the initial example, 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

```c
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)
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

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

Broadcast:

- P0 sends A to all processes.
- P1 receives A.
- P2 receives A.
- P3 receives A.

Scatter:

- P0 sends A, B, C, D to all processes.
- P1 receives B.
- P2 receives C.
- P3 receives D.

Gather:

- All processes send A, B, C, D to P0.
- P0 receives A, B, C, D.
MPI Synchronization

Synchronization is provided

- implicitly by
  - Blocking communication
  - Collective communication (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.
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.

![Diagram](Stout and Jablonowski – p. 77/250)
The MPI 3.0 (September 2012) and MPI-3.1 (June 2015) standards supply advanced features like:

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

```c
MPI_Ibcast(buf, count, type, root, comm,&req);
... // compute
MPI_Wait (&req, &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 Model (SHM)** Allows sharing of single objects (e.g. arrays or data structures), supports multicore and hybrid programming
MPI Summary

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 more advanced techniques appearing in the afternoon.
Unbalanced Load

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.*

![Graph showing computational dependencies](image)
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.

\[
\text{do } i=1,n \\
V[i] = V[i] - 2*V[i-1] \\
\text{enddo}
\]

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

\[
V
\]

- Must be computed before race
- This can be computed
Forward Dependency

\[
\text{do } i=1,n \\
\quad V[i]=V[i] - 2*V[i+1] \\
\text{enddo}
\]
Forward Dependency

\[
\text{do } i=1,n \\
\quad V[i]=V[i] - 2*V[i+1] \\
\text{enddo}
\]

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**

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

```
+  +  +  +  +
+  +  +  +  +
+  +  +  +
```
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.
We start with static decompositions of the problem, with dynamic decompositions discussed in the afternoon.

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

```
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```

Minimizing Number of Neighbors

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```
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

Distributed Array

-1 0 9 10

Processor 0

-1 0 9 10
ghost
(if needed)
Processor 1

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

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

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

to send to logical | send to MPI_COMM_RANK
--- | ---
Right (my\_row, my\_col +1) | i + 1
Left (my\_row, my\_col -1) | i - 1
Up (my\_row -1, my\_col) | i - \sqrt{p}
Down (my\_row +1, my\_col) | i + \sqrt{p}

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

Very general graph decomposition techniques can be used when communication pattern is less regular.

- Objects (calculations) represented as vertices (with weights if calculation requirements uneven)
- Communication represented as edges (with weights if communication requirements 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 tools, such as Metis, available. Typically used via off-line preprocessing.
  [http://glaros.dtc.umn.edu/gkhome/metis/metis/overview](http://glaros.dtc.umn.edu/gkhome/metis/metis/overview)
- Parallel version, ParMetis, also available. Used during program execution.
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 (e.g., size)
- May use simple measurements on small test cases.

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

For an implementation, see the references.
The best general-purpose geometric load-balancing comes from space-filling curves.

The Hilbert Space-Filling Curve
Space-Filling Curves

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

The Hilbert Space-Filling Curve
Using A Space-Filling Curve

Letters represent work, boldface twice as much work.

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 1: Determine Space-Filling Coordinates
Step 2: Sort by Space-Filling Coordinates

\[
\begin{array}{cccc}
A & B & C & D \\
E & F \\
G & H & I & J \\
K & L & M \\
N & O & P \\
Q & R & S \\
T & U & V & W \\
X & Y \\
Z \\
\end{array}
\]

\[
\begin{array}{cccc}
A & B & C & D \\
63 & 49 & 48 & 47 \\
E & F \\
50 & 40 \\
G & H & I & J \\
K & L & M \\
N & O & P \\
Q & R & S \\
T & U & V & W \\
X & Y \\
Z \\
\end{array}
\]

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>49</td>
<td>48</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>56</td>
<td>55</td>
<td>52</td>
<td>34</td>
<td>39</td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>O</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>53</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>31</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

|   | T | U | V | W |
|---|---|---|---|
| 8  | 11 | 24 | 25 |

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

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

A.k.a. Morton or shuffled bit ordering. For 2-D, point \((x_2x_1x_0, y_2y_1y_0)\) mapped to \(y_2x_2y_1x_1y_0x_0\)

For 3-D, \((x_k \ldots x_1x_0, y_k \ldots y_1y_0, z_k \ldots z_1z_0)\) \(\rightarrow z_k y_k x_k \ldots z_1 y_1 x_1 z_0 y_0 x_0\)
Which SFC to Use?

- Both extend to arbitrary dimensions.
- 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.
Shared Memory Parallelization

Shared memory (SM) machines have always been important in high performance computing.

- All processors can directly access all the memory in the system (though access time can be different).
- This greatly reduces communication latency.
- However, synchronization errors can be quite subtle.
Since 1997: OpenMP is the industry standard for shared memory programming.

In 11/2015: The OpenMP Version 4.5 specification was released (advanced feature: accelerator support).

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 (directives)

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 (embedded 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
OpenMP: Work-sharing Constructs

- **DO/for loop WORKSHARE**
  - Fork
    - DO/WORKSHARE
    - Join
    - master thread
  - team
- **SECTIONS**
  - Fork
    - Join
    - master thread
  - team
- **SINGLE**
  - Fork
    - Join
    - master thread

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

Stout and Jablonowski – p. 119/250
OpenMP Barrier

- Barriers maybe needed for correctness
- Synchronization degrades performance, avoid if possible

*Source: R. van der Pas, Overview of OpenMP 3.0*
*twomp.zih.tu-dresden.de/downloads/2.Overview_OpenMP.pdf*
OpenMP Notation and Parallel Loops

OpenMP recognizes compiler directives that start with

$OMP \quad \text{(in Fortran)}$

#pragma omp \quad \text{(in C/C++)}

⇒ Fortran Example: Parallel loop in a parallel region

!$OMP$ 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:

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

- 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
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.
Quiz: Is there something wrong?

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

! 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:

\[
\text{DO } i = 1, n \\
\quad A(\text{pid}) = A(\text{pid}) + B(i) \\
\text{END DO}
\]

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

Performance is horrible!
False Sharing: Be Aware of the Cache
False Sharing

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!
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. 137/250
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?
In this part we examine hybrid and accelerator computing models and consider tools to help develop efficient parallel programs. We also discuss data intensive computing and more advanced aspects of load balancing.

We conclude with some comments about using parallel systems, and a review.
Big data is poorly defined, often described in terms of the 3 v’s: volume, velocity, variety.

Other important aspects are veracity, complexity.

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”

- A very common architecture
- Shared disk system often managed separately, may have multiple systems attached
- For some data analysis tasks, adding massive shared memory gives big improvement over shared disk.
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: https://hadoop.apache.org

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

Map Reduce

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.

- More complex analytics often involves
  - filtering or some other map-like operation, and
  - more intensive calculations, often involving irregular structures such as graphs.
In-Memory Analytics

- One downside of Hadoop is the extensive data movement, especially for more complex analytics.

- There is rapid growth of in-memory systems such as open source Spark. [http://spark.apache.org](http://spark.apache.org)

- These tend to provide more advanced analytical tools, along with significantly higher performance when the data can be held in memory.

- Some vendors (such as Cray and IBM) have large compute systems aimed at such uses.

- These expensive systems tend to have very large memory, perhaps with extensive solid state disk capacity.
Most parallel computers employ both shared memory (SM) and distributed memory (DM) components. These machines are so-called hybrid computers.

Hybrid architectures are becoming even more complicated: heterogeneous machines with accelerators like Graphics Processing Units (GPUs).

The hybrid programming model combines shared and distributed memory programming (e.g. a selected combination of MPI, OpenMP, CUDA, OpenCL or OpenACC directives).

Advanced tutorial on hybrid programming:

https://fs.hlrs.de/projects/rabenseifner/publ/sc13_tut123_hybrid_v02.pdf
SMP node: Several multi-core CPUs are typical, OpenMP is popular parallelization choice on node

Network communication (e.g. MPI) required to move data from one SMP node to another

[Diagram showing Hybrid Memory Architecture]
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
**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 left due to cache & load-balancing aspects, but more communication.
Hybrid Programming Strategy

Many possibilities MPI+X:

http://www.hpcwire.com/2014/07/16/compilers-mpix/

- Popular choice MPI+OpenMP:
  - Often: One MPI process per SMP node
  - Only let the master thread (serial OpenMP part) send/receive MPI messages

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

- Example on the next page:
  - Use domain decomposition (for MPI parallelization)
  - In each domain: often big outer loop, e.g. over time
  - Inside the big loop: more loops over e.g. space
  - Parallelize the outermost space loop with OpenMP

Stout and Jablonowski – p. 156/250
First Implementation (in C notation)

```c
for (...) { /* time loop, serial in each MPI domain */
    // Initialization
    { ... }

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

    // Communication /* by a single thread */
    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
We’ll continue the discussion of load-balancing, looking at some more complicated problems.
Given digital image of an island, determine the types of vegetation. Easy load-balance for 16 processors:
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

Partition image into multiple pieces and assign each processor a subsquare in each piece.
How Much Scattering?

More pieces:

\[ \downarrow \text{load imbalance, i.e., } \downarrow \text{calculation time} \]
\[ \uparrow \text{overhead and/or communication time} \]

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.
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: __________
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.
**Example: Dynamic Geometry**

*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.

http://www.eecs.umich.edu/~qstout/abs/SC97.html
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.
Dynamic Load-balancing

Metis

Space Filling Curve

Movie

Movie

 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 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).
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)  (spouse/you)

Distributed Memory: manager is a processor
Shared Memory: “manager” is a shared data structure
**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</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</td>
</tr>
<tr>
<td>5</td>
<td>e i i m p t t</td>
</tr>
<tr>
<td>6</td>
<td>f f f f q 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>1</td>
<td>a a a a a a s s s e o</td>
</tr>
<tr>
<td>2</td>
<td>k k k k k k t 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 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 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

- However, queuing imposes communication overhead.

- Yet another tradeoff: cost of moving task versus cost of load imbalance when 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.

- "Chunk" tasks to reduce overhead.
  - Example: assigning loop iterations to cores in OpenMP: SCHEDULE(DYNAMIC,chunk)
  - Might use large chunks initially, then decrease them near the end to fine-tune load balance: SCHEDULE(GUIDED,chunk)

- Distributed memory: use distributed queues. 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 Summary

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

Graphics Processors Become Versatile
**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)
Accelerators Gain Popularity

Source: top500.org
Heterogeneous Architectures

- Trend: **Highly diverse** computing platforms may include multi-cores, SMP nodes, and accelerator co-processors.

- First 3 systems on Top500 6/2016 list use accelerators:
  - #1 Sunway TaihuLight, Sunway chip, similar to Intel Xeon Phi
  - #2 Tianhe-2 (Milky Way 2), Intel Xeon Phi
  - #3 Titan, NVIDIA Kepler GPU

- Microsoft Bing uses field programmable gate array (FPGA) accelerators specialized for search

- Caveat: Very difficult to use heterogeneous hardware effectively.
Graphics Processing Units (GPUs)

Variants of graphics rendering devices for a personal computer or game console have become General Purpose GPUs (GPGPUs) used as co-processors in high-performance computing.

- **Cheap**: GPUs produced in the millions, though GPGPUs are more expensive.
- **A parallel many-core architecture**, details vary widely.
- **Combination of SIMD fine-grain parallelism and slightly coarser grained MIMD**
- **Highly parallel structure makes them more effective than general-purpose CPUs for some algorithms.**
Examples of Accelerators

- NVIDIA Pascal
  - 5 TFlops double precision
  - Also has 1/2 precision!
  - 1584 cores
  - 720 GB/s memory bandwidth, 16 GB RAM

- Intel Xeon Phi Knight’s Landing
  - 3 TFlops double precision
  - 72 cores (288 threads)
  - 384 GB/s memory bandwidth, 16 GB RAM
NVIDIA Pascal Architecture

- SIMT: Single Instruction, Multiple Threads, very similar to SIMD
- Thread blocks executed by “warp” of 32 streaming processors
- Performance highly dependent on memory alignment and eliminating shared memory access
Far fewer cores, each powerful enough to function on its own, e.g., utilize MPI

Cores include a 512 bit SIMD vector processing unit, capable of 16 SP or 8 DP operations simultaneously

Separate L2 caches but kept cache-coherent
Programming an Accelerator

- **Low-level programming model:** Major code re-write
  - Proprietary programming languages or extensions, such as NVIDIA’s CUDA
  - Portable: OpenCL (Open Computing Language)
    - [http://www.khronos.org/opencl/](http://www.khronos.org/opencl/)

- **High-level programming model:** Compiler directives
  - OpenMP 4.5 support for accelerators
  - Single code base: Compile same program on multi-core CPU or CPU/GPU systems, portable
CUDA Example

Given arrays $a, b$ compute $c = \sum_{0}^{n-1} a(i) \times b(i)$

CPU

- $a$
- $b$
- $c$

CUDA Copy Host To Device

GPU

- $a$
- $b$
- $c$
- $\text{temp}$

CUDA Copy Device To Host

- GlobalIdx = threadIdx.x + blockIdx.x * blockDim.x
- $\text{temp}[\text{threadIdx.x}] = a[\text{globalIdx}] \times b[\text{globalIdx}]$
- $\text{syncthreads}$
- If $(0 == \text{threadIdx.x})$ sum $\text{temp}$ within block
- $\text{atomicAdd}(c, \text{sum})$

All threads active

1 thread/block uses block SM

Serial, blocks don't share mem
int main(void) {
    int *a, *b, *c; // CPU copies
    int *dev_a, *dev_b, *dev_c; // GPU copies
    arraysize = n*sizeof(int);
    cudaMalloc( (void**) &dev_a, arraysize);
    cudaMalloc( (void**) &dev_b, arraysize);
    cudaMalloc( (void**) &dev_c, sizeof(in));
    use malloc for a, b, c

    initialize a, b
    cudaMemcpy(dev_a, a, arraysize, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, arraysize, cudaMemcpyHostToDevice);

    // launch GPU kernal
    dot <<< n/threads_per_block,threads_per_block >>>(dev_a,dev_b,dev_c);

    //copy results back
    cudaMemcpy(c, dev_c, sizeof(int), cudaMemcpyDeviceToHost)
CUDA Example: GPU Code

```c
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void dot (int *a, int*b, int*c) {
    int temp[threads_per_block];
    int index = threadIdx.x + blockIdx.x * blockDim.x;  // convert warp index to global
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();  // avoid race condition on sum within block

    if ( 0 == threadIdx.x) {  // thread 0 computes sum within block
        int sum = 0;
        for (int i = 0; i<threads_per_block; i++)
            sum += temp[i];
        atomicAdd(c, sum);    //add to global sum
        //use atomic to avoid interleaved read/writes
    }
}

Example borrowed from J.S.Vetter,

http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2012-02-20/07-intro_to_cuda_c.pdf
```
OpenMP and OpenACC share a model of computation

- **Host-centric**: host device “offloads” code regions and data to accelerators for execution
- **Device**: (execution engine)
  - has independent memory
  - has multiple threads organized in some fashion
- **Mapping** clause: defines the relationship between memory on the CPU and that on the device
OpenMP Accelerator Example

- Compute \( A(i) = A(i) \times B(i) + 1 \) for arrays A and B
- This code works well on Intel Xeon Phi

```c
... initialize arrays A, B
#pragma omp target device(0) map(tofrom: A) map(to: B)
#pragma omp parallel for
for (i=0; i<N; i++) {
}
```

- Environment Variable OMP_DEFAULT_DEVICE=<int>: sets device number for target constructs
  - Here: Device 0 is the accelerator
- Array A copied from host to accelerator and back to host
- Array B only copied from host to accelerator
How Much Speedup Can Be Expected?

- Some problems achieve speedup of 10X-100X (e.g., Linpack benchmark)

- However:
  - only a fraction of most applications can utilize the GPU effectively
  - many applications show speedups around 2X – 4X
  - often multi-core CPU code can be further optimized, this may be your best use of time

There are several problem areas that lower performance:
Branch Divergence Slows SIMD

- On SIMD systems

  for i=0, n
    if X(i) > 0 then
      X(i) = X(i) + C*Y(i)
    else
      X(i) = Z(i)**2
    end

decreases utilization by 50% since system needs to issue instructions for both branches

- SIMT (Single Instruction, Multiple Thread) is SIMD execution, central to NVIDIA and AMD systems
Memory Bandwidth Bottleneck

- GPUs have limited off-chip memory access bandwidth compared to peak compute throughput

- Example: NVIDIA’s Pascal
  - 5.3 TFlop/s peak double precision (DP)
  - 720 GB/s peak off-chip memory access bandwidth
  - Max 720 GB/s / 8 = 90 G DP operands/sec
  - To achieve peak FLOPS requires $5300 / 90 \approx 59$ DP floating point arithmetic operations for each operand value fetched from off-chip memory
  - If operands only used once peak is 90 GFlops (DP), may be much less if memory not aligned optimally
Ideally, we want

In reality, we might get
Accelerator Trends

- GPGPUs becoming more flexible, less rigidly SIMD
- Bandwidth is improving, e.g., direct memory access eliminates offloading of the data from the host to GPU
- Progress in going from low-level programming (CUDA, OpenCL) to higher level directives (OpenMP, OpenACC)
  
  However:
  - NVIDIA bought Portland Group in July 2013, PGI compilers stopped supporting OpenCL
  - OpenACC and OpenMP haven’t merged

- Overall, programming model is a moving target, efficiency still problematic for many applications
New Motivator: Machine Learning

- Several machine learning tasks can utilize accelerators efficiently.
- Deep learning example: convolutional neural net to classify an image

Source: adespande3.github.io
Convolution Step For Finding Birds

Given a small pattern, find everywhere it occurs in image

- The $4 \times 4$ pattern can be compared in parallel to subsquares throughout the image
- Perfect for GPU
Neural Learning Levels

Calculating $\sum$ weighted inputs for all neurons at a level is matrix multiplication.

GPUs can do this very efficiently.
Rapidly Evolving Hardware Support

- Deep learning usage growing exponentially (search, audio processing, autonomous vehicles, ...)
- Different computational requirements than regular HPC
  - Neural nets require far less precision
  - More operations/second at lower energy/operation
  - This is why NVIDIA’s Pascal has 1/2 precision
- Google developed Tensor Processing Unit chip for this, other companies similarly developing special chips
- Facebook open sourced “Big Sur” AI server design
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 MFlop/s, GFlop/s, TFlop/s or PFlop/s.
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
Almost factor $\approx 10^6$ increase over the last 23 years
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

\[
\begin{align*}
do & \ i=1, n \\
do & \ j=1, n \\
enddo
\end{align*}
\]

\[
\begin{align*}
do & \ j=1, n \\
do & \ i=1, n \\
enddo
\end{align*}
\]

For large arrays, which is faster, and why?
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 \textit{not the default}. Thus

\textbf{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.

\textbf{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 (memory, overhead)?
  - Do you have access? Commercial versus public domain tools
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 Flop/s 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 5.4.3 released in 1/2016
- 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, OpenSpeedShop

Home page:
http://icl.cs.utk.edu/papi/
Profiling/Analyzing 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 or Performance Reports
(www.allinea.com)

Public domain performance tool by the Krell Institute
Open|SpeedShop: http://www.openspeedshop.org
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
- Public domain tool **Open|SpeedShop**: http://www.openspeedshop.org


**Vampir** and **Vampirtrace** (commercial, all platforms)

Trace analyzer developed by the Center for Information Services and High Performance Computing, Dresden, Germany (http://www.vampir.eu)
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
Performance data (GFlop/s) via hardware monitor PAPI
Communication pattern and message sizes
Performance Analysis: Strategy

- **Hardware counters/Vampir** provide information on Flop/s 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: https://computing.llnl.gov/tutorials/totalview/
TotalView: The Process Window

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

http://www.eclipse.org/ptp/
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](https://www.xsede.org)

Another source of large allocations: DOE INCITE  
[http://www.doeleadershipcomputing.org/incite-program/](http://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](http://www.prace-ri.eu)

**Businesses:** Can purchase time from vendors: Amazon Web Services, Google Cloud Platform, Microsoft Azure, etc., sometimes from national or university centers.
We’ll 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.

Projected Performance Development

Original source: https://www.top500.org/statistics/perfdevel/ with additional annotations
Chip Trends

Energy Constrains Options

Problem: largest systems use > $10M electricity/year
Need nuclear reactor for *Exascale*: $10^{18}$ flops

Hardware Solutions (?)
- Power $\approx$ speed$^2$, so lower clock speed and use more cores to regain flops.
- Reduce RAM/core since RAM also uses power.
  - Sunway TaihuLight: 10,649,600 cores, 0.1 GB RAM/core
  - Sustained Exascale ($\approx$ 2023): $10^9$ cores, ? GB/core

*Tradeoffs opposite programmer needs*
- Amdahl’s law shows want fewer, faster, cores
- Many applications need better resolution: increases # time steps, which are serial, need faster cores

Stout and Jablonowski – p. 238/250
An Approach to Exascale

100,000 brains

10^{18} \text{ ops/sec}

2 M Watt

Potential alternative: *neuromorphic computing*
More Trends

- Accelerator-based machines grab top rankings, but clusters with standard cores are most important economically.

- $/flop continues to plummet: for 1 Gflop
  
  1961: $\approx 1,100,000,000,000$. Now: $\approx 0.08$ (GPU)


- Irregular data access, such as for data analytics, increasingly important.
  Graph500, RIKEN K computer is # 1 (# 5 on Top500)

  [http://www.graph500.org](http://www.graph500.org)
Systems Have Many Layers

Economics dictates distributed memory system of shared memory boards with multicore commodity chips, perhaps with some form of accelerator.

- Mixed mode programming complex to write and optimize
- Need smarter compilers and run-time systems
- Need new approaches and better algorithms
MPI + X

For the foreseeable future, efficient, practical programming will use MPI + X (or MPI + X + Y).

Other options include Chapel and X10
If It Isn’t Working Well . . .

Early approaches probably weren’t developed for parallelism

Stout and Jablonowski – p. 243/250
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.

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.
RESOURCES and REFERENCES

Selected web resources for parallel computing are (occasionally) maintained at

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


Amdahl’s short paper covers several aspects of serial vs. parallel computing, and the equation is only implied, not explicitly written out.


Graph500: see Rankings

Green500: see Rankings

Hadoop: http://hadoop.apache.org

Hilbert space-filling curve: see the routines available in Zoltan (listed below), or use a web search to find various implementations.

Metis & Parmetis: http://glaros.dtc.umn.edu/gkhome/views/metis

Free, portable versions at:


OpenACC: [http://www.openacc.org](http://www.openacc.org)

OpenCL: [http://www.khronos.org/opencl/](http://www.khronos.org/opencl/)


Parallel computing, a slightly whimsical explanation

[http://www.eecs.umich.edu/~qstout/parallel.html](http://www.eecs.umich.edu/~qstout/parallel.html)

Rankings of supercomputers

- **Top500**: [http://www.Top500.org](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](http://www.green500.org), uses rankings based on flops/watt, a metric which is increasingly important.

- **Graph500**: [http://www.graph500.org](http://www.graph500.org), uses performance on sparse graph problems, a benchmark more suitable for problems such as data analytics.
Spark: [http://spark.apache.org](http://spark.apache.org)

Top500: see Rankings.

Training material: an extensive collection of good classes for HPC
[https://computing.llnl.gov/?set=training&page=index](https://computing.llnl.gov/?set=training&page=index)

UPC (Unified Parallel C): [http://upc.gwu.edu](http://upc.gwu.edu).

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)

Zoltan, a collection of routines for load balancing et al., including space-filling curves.