Organization

Part I Introduction, Terminology
Example (crash simulation)
Speedup and Efficiency, Amdahl’s Law
Architectures
Distributed Memory Communication, MPI
Parallelizing Serial Programs I
Load Balancing I
Shared Memory, OpenMP
Part II  Hybrid Computing
Pipelining and GPUs
Parallelizing Serial Programs II
Load Balancing II
Data Intensive Computing
Performance Improvement, Tools
Using Parallel Systems
Review, Wrapup
In this part 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
  - Kilo-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, ...

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 and access to all resources.

**Multi-core Processors**  Multiple processors (cores) on a single chip. Aka many-core. Heterogeneous multi-core chips with GPU being developed.

**Cluster Computing**  Hierarchical combination of commodity units (processors or SMPs) to build parallel system.
**Supercomputing**  Use of the fastest, biggest machines to solve large problems. Historically vector computers, but now are parallel or parallel/vector.

**High Performance Computing**  Solving problems via supercomputers + fast networks + visualization.

**Pipelining**  Breaking a task into steps performed by different units, with inputs streaming through, much like an assembly line.

**Vector Computer**  Operation such as multiply broken into several steps and applied to a stream of operands (pipelining with “vectors”).
Pipelining, Detroit Style
Who Uses Supercomputers?

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

From the Top 500 list as of June. The newest list has just been announced: http://www.top500.org
A greatly simplified model, based on parallelizing crash simulation for Ford Motor Company. Such simulations save a significant amount of 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 to new codes developed for parallel computers.
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
Software Engineering Aspects

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.

Life-cycle costs are often overlooked until it is too late!

Note that high-level structure same as serial version: a sequence of steps. The sequence is a serial construct, but steps are performed in parallel.
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.
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.
Ghost Cells
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.
Coding and Correctness?

How does one manage the software engineering of the parallelization process?

- Utilize an incremental parallelization approach.
- Constantly check test cases to make sure answers correct.
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

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

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

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

Speedup: \( \frac{\text{SerTime}(n)}{\text{ParTime}(n,p)} \)

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

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

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

Linear speedup: speedup = \( p \).
Technically, is linear if speedup \( \geq c \cdot p \) for some \( c \).

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

Number of Processors

Perfect
Common
Occasional

Speedup

Number of Processors
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 [1967]: Let \( f \) be fraction of time spent on operations that are performed serially. Then for \( p \) processors,

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

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

\[
\text{Speedup}(p) \leq \frac{1}{f + (1 - f)/p}
\]

Thus no matter how many processors are used:

\[
\text{Speedup} \leq \frac{1}{f}
\]
Unfortunately, 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.
Maximal Possible Speedup

![Graph showing speedup vs. number of processors for different values of f (0.001, 0.01, 0.1).]

- **f=0.001**
- **f=0.01**
- **f=0.1**

Stout and Jablonowski – p. 33/284
Parallelization usually adds work, typically communication, which reduces speedup.

For example, crash simulation typically runs for a fixed simulated time interval. Due to the physics of the situation, if use $n$ finite elements, number of time steps grows like $\sqrt{n}$, so serial processor time grows like

$$C_1 \cdot n^{1.5}$$

for some $C_1 > 0$. 
Suppose use $p$ processors. Every time step processors receive and send information about border elements. There is also periodic global communication of total energy, contact, etc.

For simple approaches, communication time grows like

$$\sqrt{n} \left( C_2 \cdot p + C_3 \sqrt{n/p} \right), \quad C_2, C_3 > 0$$
Effect of Communication

Suppose $C_2 = C_1 = 10$ and $C_3 = 1$. Then for $n = 1000$ we get the following speedup.
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

- 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 \( n \)

Sometimes serial portions grow with problem size but much slower than the total time. I.e., Amdahl’s “f” decreases as \( n \) increases.
Scaling

For such programs, can often exploit large parallel machines by scaling the problems to larger instances.

To illustrate, use a model like the crash simulation

\[ SerTime(n) = 10 \cdot n^{1.5} \]

and the time for \( p \) parallel processors grows like

\[ ParTime(n, p) = 10 \cdot n^{1.5} / p + 10 \cdot p\sqrt{n} + n / \sqrt{p} \]
Fixed Size per Processor

Fixing the amount of data per processor usually gives highest efficiency possible, hence it is commonly cited. Called **weak scaling**.

Suppose each processor can hold 1000 elements.
Fixed Time


Fix time to be $\text{SerTime}(1000)$.
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 are in terms of hardware, but there are natural software analogues.

Note: 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

\[
\left\{ \begin{array}{c}
S \\
M
\end{array} \right\} \quad I \quad \left\{ \begin{array}{c}
S \\
M
\end{array} \right\} \quad D
\]

**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 is rare — some extreme fault-tolerance schemes, using different computers and programs to operate on same input data, are of this type.

Almost all parallel computers are MIMD.

SIMD: there used to be companies that made such systems (Thinking Machines’ Connection Machine was the most famous).

Vector computing is a form of SIMD.
A SIMD System

Controller, with program

Instructions

Processors, with data
Data parallel software — 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: a coarse-grained SIMD approach to programming for MIMD systems.
Memory Systems: Distributed Memory

- All memory is associated with processors.
- If processor $A$ needs data in processor $B$, then $B$ must send a message to $A$ containing the data.

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

Stout and Jablonowski – p. 49/284
Disadvantages

- Programmer is responsible for many of the details of the communication, easy to make mistakes.

- May be difficult to distribute the data structures, often need to revise them to add additional pointers.
Memory Systems: Shared Memory

- Global memory space, accessible by all processors
- Processors may have local memory to hold copies of some global memory.
- Consistency of copies is usually maintained by hardware.

Advantages:
- Global address space is user-friendly, program may be able to use global data structures efficiently and with little modification.
- Data sharing between tasks is fast
Disadvantages

- System may suffer from lack of scalability. Adding CPUs increases traffic on shared memory - to - CPU path. This is especially true for cache coherent systems.

- Programmer is responsible for correct synchronization.

- Systems larger than an SMP need some special-purpose components.
Shared vs. Distributed

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

**DISTRIBUTED MEMORY**
- Network
  - Processor + Cache + Memory
Shared Memory Access Time

Two classes of SM systems based on memory access time:

**Uniform Memory Access (UMA):**
- Most commonly represented by Symmetric Multi-processor Machines (SMP), identical processors
- Equal access times to memory
- Some systems are CC-UMA (cache coherent UMA): if one processor updates a variable in shared memory, all the other processors know about the update.
SM Access Time continued

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP (not message-passing)
- Memory access times are not uniform, memory access across a link is slower
- Cache coherent systems: CC-NUMA
Shared Memory on Distributed Memory

As we’ll see later, it is usually easier parallelize a program on a shared memory system.

However, most systems are distributed memory because of the cost advantages.

To gain both advantages people have investigated virtual shared memory, or global address space (GAS), using software to simulate shared memory access.

Current projects include Unified Parallel C (UPC) and Co-Array Fortran.
Virtual Shared Memory Performance

Communication time in distributed memory machines is quite high compared to local RAM access. Thus virtual shared memory access time is highly nonuniform.

Because of these access delays, the performance of these systems is not good, even if reasonable care is taken, but may be justified by greatly reduced programmer time.

*Software and hardware models need not match, though there are often performance problems when they don’t.*
Communication Network

There are many ways that the processors can be interconnected but for the user the 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**  Processors (and memory) connected to routing switches like in telephone system.
Networks

Switch

Processor

Multistage Interconnect

Bus
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 needs to ensure that different caches don’t contain different values for the same memory locations (cache coherency). This is 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.
The mismatch of processor speed and memory speed causes a bottleneck. There is an inverse relationship between memory speed and $/\text{byte}$ and there are physical constraints on the size of memory.
## Speed-Size Tradeoff

<table>
<thead>
<tr>
<th>Speed</th>
<th>Time Unit</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>nanosec</td>
<td>MByte</td>
</tr>
<tr>
<td>Ram</td>
<td>100 nanosec</td>
<td>GByte</td>
</tr>
<tr>
<td>Disk</td>
<td>10 millisec</td>
<td>100 GByte</td>
</tr>
<tr>
<td>Tape</td>
<td>minute</td>
<td>100 TByte</td>
</tr>
</tbody>
</table>

Data is moved between levels in blocks (cache lines, pages). Effective high-performance computing includes arranging data and program so that entire block is used while resident in the faster memory.
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.
Measured Performance

For example, a 4.7 GHz IBM Power 6 (p575) processor, best case MPI messages (discussed later):

- processor speed: 4700 cycles per microsecond ($\mu$sec), 4 flops/cycle, 18800 flops per $\mu$sec.

- MPI message latency, caused by software:
  \[ \approx 1.3 \ \mu \text{sec} = 24,400 \text{ flops} \]

- message bandwidth, usually limited by hardware:
  \[ \approx 2500 \text{ bytes per } \mu \text{sec} = 7.5 \text{ flops/byte} \]

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.
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.
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-unix.mcs.anl.gov/mpi
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 and Fortran.
- Supplies many communication variations and optimized functions for a wide range of needs.
- Supports large program development and integration of multiple modules.
- Many powerful packages and tools based on MPI.
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
http://www.llnl.gov/computing/tutorials/mpi/

Basic and advanced MPI tutorials, e.g. on I/O and one-sided communication
http://www-unix.mcs.anl.gov/mpi/tutorial/
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
http://www.llnl.gov/computing/tutorials/mpi/

Basic and advanced MPI tutorials, e.g. on I/O and one-sided communication
http://www-unix.mcs.anl.gov/mpi/tutorial/

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 system can provide the overlap.
MPI Initialization

Near the beginning of the program, include

```
#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.
MPI Example

Each processor sends value to proc. 0, which adds them.
Basic Program

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

```c
MPI_Recv(&value, 1, MPI_FLOA T,
    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 Collective 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.

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.
Collective Communication

Broadcast

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

Scatter

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

Gather

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>9</th>
<th>13</th>
<th>17</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

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

```c
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
The MPI-2.2 standard was just approved by the MPI Forum on September 4, 2009, updates the MPI-2.0 standard from 1997. Important added features in MPI-2.x include

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 these are are the same as directly accessing remote memory.

However, these are risky and can easily introduce race conditions.
One-Sided Communication

![Diagram showing one-sided communication between processors and memories.](image-url)
MPI Summary

The MPI standard includes

- point-to-point message-passing
- collective communications
- group and communicator concepts
- process topologies (e.g. graphs)
- environmental management (e.g. timers, error handling)
- process creation and management
- one-sided communications
- parallel I/O routines
- profiling interface
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.
Serial Performance

Often profiling reveals serial performance problems — eliminating these may be critical to attaining performance goals.

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

Set Goals

Analyze, profile

Prioritize Changes

Incrementally change

Verify correctness

Performance acceptable?

Yes

Code ready for use

No
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. This is a major advantage of shared memory over distributed memory.

Some benefits to this approach:

- Smaller changes make it is easier to locate mistakes.
- It is easier to determine where efficiency is poor.

Should have test cases available and constantly verify correctness.
Process for Distributed Memory

While more complicated, an incremental approach can also be utilized for distributed memory machines.

It is harder to get started, but the basic approaches are similar. The first things one needs to do are

- Do coarse-grained profiling, to determine the time consumed in the different sections of the program.
- Develop maps of the major data structures and where they are used.

The profiling is used to prioritize the areas that need to be parallelized.
Parallelization Steps

Once parallelization plan ready, start parallelizing sections of code and data structures.

- 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.

- Profile to see if efficiency of this piece is acceptable. If not, then develop better alternative.
Incremental DM Parallelization
Useful to retain the serial-parallel scaffolding (normally turned off), to help maintain the correspondence between the serial and parallel codes as they evolve.

This is probably a complex, important program, since it is worth parallelization effort. Therefore software engineering concerns, such as life-cycle maintenance, are very important.
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.
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.*

**Computational Dependencies**

![Diagram of Computational Dependencies](image_url)
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.

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

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

\[
V
\]

Must be computed before This can be computed
Forward Dependency

To parallelize

do i=1,n
  V[i]=V[i] − 2*V[i+1]
enddo

Race

Must be computed before

This is updated

V
Copy V and use copy to compute new values.

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

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**

\[
y[0] = a[0] \\
\text{for } i = 1, n-1 \\
\quad y[i] = y[i-1] + a[i] \\
\text{end for}
\]
Reduction and scan operations are extremely common.

They are recognized by parallelizing compilers and implemented in MPI and OpenMP.
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.
Complicating Factors

- Objects being computed may not have a simple dependency pattern among themselves, making communication load-balancing difficult to achieve.

- Objects may not have uniform computational requirements, and it may not initially be clear which ones need more time.

- If objects are repeatedly updated (such as the elements in the crash simulation), the computational load of an object may vary over iterations.

- Objects may be created dynamically and in an unpredictable manner, complicating both computational and communicational load balance.
Static Decompositions

Here we will consider only static decompositions of the work, with dynamic decompositions discussed later. A variety of basic techniques available, each suitable for a different range of problems.

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.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimizing Boundary</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimizing Number of Neighbors</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
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

Processor 0

Processor 1

Processor 2

ghost (if needed)
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 $n_{local} = n/p$

- $A[i]$ on processor $p$ corresponds to $A[i + p \times n_{local}]$ in the original array

- if use $A[i+1]$ and $A[i-1]$ in calculation of $A[i]$, $(i \neq 0, n-1)$, then would have $A[−1 : n_{local}]$ to add ghost cells
Linear Rank vs. 2-D Indices

To map processor ranks 0..15 to rows 0..3 and columns 0..3

For processor rank $i$, \( \text{row}_i = \lfloor i / \sqrt{p} \rfloor \) and \( \text{col}_i = i - \text{row}_i \times \sqrt{p} \)

- **Right:** \((\text{row}_i, \text{col}_i + 1), \text{rank } i + 1\)
- **Left:** \((\text{row}_i, \text{col}_i - 1), \text{rank } i - 1\)
- **Up:** \((\text{row}_i - 1, \text{col}_i), \text{rank } i - \sqrt{p}\)
- **Down:** \((\text{row}_i + 1, \text{col}_i), \text{rank } i + \sqrt{p}\)

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

Very general graph decomposition techniques can be used when communication patterns 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

Unfortunately, optimal graph decomposition is NP-hard.

Fortunately, various heuristics work well, and high-quality decomposition tools are available, such as Metis.

To use a serial tool such as Metis, convert data into format it requires, run Metis to partition the graph vertices, then convert to format your program requires.

Scripts (Perl, Python, etc.) useful to convert formats.

Parallel version, ParMetis, also available.

http://www.cs.umn.edu/~karypis/metis/metis.html
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.
The best general-purpose geometric load-balancing comes from space-filling curves.

The order in which points are visited in the space-filling curve determines how the geometric objects are grouped together to be assigned to the processors.
The Hilbert Space-Filling Curve

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

```
A B C D E F G
63 49 48 47 40 50 59
H I J K L M N
56 55 52 34 39 38 36
O P Q R S T U
53 58 6 31 27 25 24
V W X Y Z
29 8 2 23 19
```
Step 2: Sort by Space-Filling Coordinates

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

A B C D E F G
63 49 48 47 40 50 59

H I J K L M N
56 55 52 34 39 38 36

O P Q R S T U
53 58 6 31 27 25 24

V W X Y Z
29 8 2 23 19

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

Aka 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, \((x_k \ldots x_1x_0, y_k \ldots y_1y_0, z_k \ldots z_1z_0) \rightarrow z_ky_kx_k \ldots z_1y_1x_1z_0y_0x_0\)
Hilbert vs. Z

- 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.

- Also: data parallel extensions to Fortran90, High Performance Fortran (HPF), but lack of efficiency.
Parallelization Techniques: OpenMP

- Since 1997: OpenMP is the new industry standard for shared memory programming.
- In 2008: The OpenMP Version 3.0 specification was released (new feature: task parallelism).
- 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

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
http://www.llnl.gov/computing/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 3-4 directives.

- **Ease of use**: supports incremental parallelization of a serial program, unlike MPI which typically requires an all or nothing approach.

- **Portability**: supports Fortran (77, 90, 95), C (C90, C99) and C++
OpenMP: 3 Building Blocks

Compiler directives (imbedded in user code) for
- parallel regions (PARALLEL)
- parallel loops (PARALLEL DO)
- parallel sections (PARALLEL SECTIONS)
- parallel tasks (PARALLEL TASK)
- sections to be done by only one processor (SINGLE)
- synchronization (BARRIER, CRITICAL, ATOMIC, locks, etc.)
- 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.
OpenMP: The Fork-Join Model

Parallel execution is achieved by generating \textit{threads} which are executed in parallel (multi-threaded parallelism):

\begin{itemize}
  \item \texttt{fork}:
  \begin{itemize}
    \item master thread
    \item parallel region
    \item \texttt{fork}
    \item \texttt{join}
    \item parallel region
    \item \texttt{fork}
    \item \texttt{join}
  \end{itemize}
\end{itemize}
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 loops**: type of “data parallelism”
- **SECTION**: breaks work into independent sections that are executed concurrently by a thread (“functional parallelism”), units of work are statically defined at compile time
- **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++)
OpenMP: Work-sharing Constructs

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

Source: R. van der Pas, Overview of OpenMP 3.0
https://www.zih.tu-dresden.de/downloads/2.Overview-OpenMP.pdf
Parallel Loops (1)

⇒ in Fortran notation

```fortran
$OMP PARALLEL DO
DO i = 1, n
    a(i) = b(i) + c(i)
END DO
$OMP END PARALLEL DO
```
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 Sections (1)

⇒ in Fortran notation

```fortran
!$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)

- Main change 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
Parallel Tasks (3)

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 the `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 Loops and Scope of Variables

- 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.

```plaintext
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
```

Stout and Jablonowski – p. 152/284
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

!$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 PARALLEL DO
Loop Collapsing

- 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.

! 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 \( pid = 0 \ldots P-1 \)

- Each processor runs the Fortran code:
  
  \[
  \text{DO } i = 1, n \\
  \quad A(pid) = A(pid) + B(i) \\
  \text{END DO}
  \]

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

  Performance is horrible!
False Sharing 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.
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

```c
!$OMP CRITICAL
<code-block>
|$OMP END CRITICAL
```

```c
!$OMP ORDERED
<code-block>
|$OMP END ORDERED
```

- 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
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.
Something to think about over the break

Question: How would you distribute the work in a climate model?
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.

The corresponding hybrid programming model is a combination of shared and distributed memory programming (e.g. OpenMP and MPI).

Today: hybrid architectures are dominant at the high end of computing.

Future: hybrid architectures are likely to prevail, but it might get more complicated: e.g. heterogeneous machines with GPU accelerators like IBM Roadrunner.
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:
- Maybe lack of scalability between memory and CPUs. Adding more CPUs increases traffic on shared memory - CPU path
- User is responsible for correct synchronization
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 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.
Hybrid Memory Architecture

CPU: single-core or multi-core technology possible

- Multi-core (dual- or quad-core) chips common, even in laptops
- Typical: Several multi-core chips form an SMP node.
Example of an SMP node

- IBM Power6 node with 16 chips and two processor per chip (center), copper tubes carry liquid coolant
- Flanked by 64 memory modules (DIMMs) on both sides
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 CC 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 Level Parallelism

Often: Applications have two natural levels of parallelism. Take advantage of it and exploit the shared memory parallelism by using OpenMP on an SMP node. Why?

- 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 and OpenMP
- 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**
  - Many MPI processes
  - 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
  - Can give better scalability than pure MPI or OpenMP

- Requires thread-safe MPI libraries and compilers!
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 *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
PIPELINING and GPUS

- Pipelining principle
- Classical vector computers
- General Purpose Graphics Processing Units (GPGPUs)
- Languages and compilers
Streaming/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
  ```

- Vector processors and GPUs are specialized for these floating point operations.

- Extremely fast.

Source: www.forwardlook.net/images/Photo09.jpg
Classical Vector Computers

- Worldwide: vector computers became less and less common over the last 16 years
- In 2009: NEC and Cray remain in this market
- Still powerful architecture:
  - 131 TFlop/s NEC SX-9 system (peak performance): Earth Simulator (Japan, #23 TOP500 list in 6/2009)
  - NEC SX-9: reaches 122 TFlop/s with only 1280 cores, impressive single-core performance
  - Cray XT5h (e.g. Edinborough), hybrid architecture with X2 vector processing node
- Extreme sustained performance: Earth Simulator system reaches approx. 90% of its peak performance (Linpack benchmark)
Modern Scalar Processors

- Multifunctional pipelines are also used in modern **scalar** processors ⇒ speed up execution

Examples:

- **IBM Power6 CPU**: Floating point units (FPU) which can issue a combined multiply/add
  \[ a = b \times c + c \]
  - Multi-functional hardware unit
  - In addition: data prefetch capabilities (“load pipeline”)

- **IBM BlueGene**: Double Hummer floating point unit
Graphics Processing Units (GPUs)

- Newest trend in high-performance computing.
- GPU contains a graphics pipeline, is an accelerator.
- GPUs have a parallel many-core architecture, each core capable of running thousands of threads simultaneously.
- SIMD fine-grain parallelism
- Traditionally: GPU dedicated graphics rendering device for a personal computer, workstation or game console, often integrated into motherboard.
- Highly parallel structure makes them more effective than general-purpose CPUs for some (highly specialized) algorithms.
Graphics Processors Become Versatile
Graphics Processing Units

- GPUs are cheap: accelerators are commodity plug-in co-processors, produced in the millions.

- Trend: **Highly diverse** computing platforms can include multi-cores, SMP nodes, graphics accelerators or classical vector units as co-processors for both thread-based and process-based parallelism.

- #1 computer on Top 500 list: “Roadrunner” utilizes GPUs as accelerators: IBM’s GPU **Cell** processor originally designed for the Sony Playstation 3.

- Extremely difficult to use the hardware effectively.
Accelerators are Commodity Parts

- NVIDEA’s Tesla GPU
  - 1 Tflops single precision performance
- AMD’s Firestream
  - 0.5 Tflops single precision performance
General Purpose GPUs

- Goal is to extend the range of applications, particularly to high-performance computing.
- Provide IEEE single and double precision arithmetic (double precision much slower)
- Some development of programming environments, such as NVIDIA’s CUDA (Compute Unified Device Architecture): compiler and set of development tools
- Data locality is important, otherwise performance degrades significantly
- Very fast, first 1 TFlop/s GPGPU was out in February 2008
How to Program a GPGPU

- Proprietary programming languages or extensions
  - NVIDIA: CUDA (C/C++ based)
  - AMD: StreamSDK or Brooks+ (C/C++ based)
- OpenCL (Open Computing Language)
- Major rewriting of the code required, *not portable*
- Newest technique: Portland Group Fortran and C compilers offer directives for the NVIDIA GPU
  - Accelerator directives similar to OpenMP in structure
  - User tells the compiler which parts of the code the GPU should execute

*Big question*: is it worth your time to develop GPGPU software?
Here we examine some of the more complicated aspects of successfully parallelizing large programs.

As before, the ideas also apply to developing a parallel program from scratch.
Some Problems & Solutions

Need to test all program options
Typically requires coordination with application expert, and careful analysis. There are tools to check coverage.

Problems only occur after extensive computing
Checkpoint/restart can help. Also very useful for long-term maintenance and for production runs so that work not lost if system fails during a long run.

Simple “diff” shows spurious differences
Use of IEEE arithmetic helps cross-platform comparisons. Also, by being careful can insure that the parallel and serial programs perform all calculations (such as summations) in the same order, but usually this lowers efficiency.
Performance Problems

Detailed profiling of the crash code showed that there were many places where efficiency was unacceptable.

- Often cache utilization was very poor.
- Load balance difficult due to heterogeneous elements with time-varying requirements.
- Contact adds dynamic computational and communication imbalance.
- I/O was substantial, and was often inefficient.
Profiling proceeded in stages, identifying where efficiency was too low.

For targeted section, profiled uniprocessor performance, such as cache misses.

Also profiled load imbalance and communication overhead, proceeding from smaller systems to larger ones (when needed).

Incremental approaches kept the amount of data collected at manageable reasonable levels.
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

```
! original
do i=1,n
    do j=1, n
        enddo
    enddo
! modified

```

For large arrays, which is faster, and why?
Utilizing 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.
Utilizing 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.*
LOAD-BALANCING II

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.
However ...

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

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

Especially problematic because large regions will be of one type or the other. Thus some processors will take much longer than others.
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. Suppose tasks take time \( i \) with probability \( 2^{-i} \), and there is no way to tell in advance how long the task will take.

If each processor does 1 task and then waits for all processors to complete before going on to the next, the efficiency is low. In fact, it grows as the log of the number of processors.

To improve efficiency, each processor needs to complete several tasks before synchronizing.
# Geometric Task Times

<table>
<thead>
<tr>
<th>No. Proc.</th>
<th>Efficiency 1 Task per Processor</th>
<th>Tasks/Proc. to achieve Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.8</td>
</tr>
<tr>
<td>4</td>
<td>0.57065</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>0.37193</td>
<td>30</td>
</tr>
<tr>
<td>64</td>
<td>0.27233</td>
<td>53</td>
</tr>
<tr>
<td>256</td>
<td>0.21423</td>
<td>78</td>
</tr>
<tr>
<td>1024</td>
<td>0.17647</td>
<td>103</td>
</tr>
</tbody>
</table>
## Another Example

Tasks: 1 time unit with prob 0.9, 10 units with prob. 0.1

<table>
<thead>
<tr>
<th>No. Proc.</th>
<th>Efficiency 1 Task per Processor</th>
<th>Tasks/Proc. to achieve Efficiency 0.8</th>
<th>Tasks/Proc. to achieve Efficiency 0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.46397</td>
<td>36</td>
<td>179</td>
</tr>
<tr>
<td>16</td>
<td>0.22803</td>
<td>112</td>
<td>536</td>
</tr>
<tr>
<td>64</td>
<td>0.19020</td>
<td>199</td>
<td>949</td>
</tr>
<tr>
<td>256</td>
<td>0.19000</td>
<td>291</td>
<td>1384</td>
</tr>
<tr>
<td>1024</td>
<td>0.19000</td>
<td>385</td>
<td>1824</td>
</tr>
</tbody>
</table>
Note that one can keep the efficiency high by assigning many tasks per processor before synchronizing, but the number required grows with the number of processors.

Later we’ll see a technique to improve this situation.
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.
Dynamic Graph Decomposition

One could rerun Metis at periodic intervals, or periodically measure some metric to determine if processor loads too uneven, and if so then call Metis.

However, more efficient to use the ParMetis package which runs in parallel.
**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. [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
Data distribution at model day 3:
Simple Load-balancing Strategy cont.

Data distribution at model day 12:
Dynamic Load-balancing with METIS

Movie

Courtesy of Dr. Joern Behrens, Alfred-Wegener-Institute, Bremerhaven, Germany
Dynamic Load-balancing with SFC

Movie
Courtesy of Dr. Joern Behrens, Alfred-Wegener-Institute, Bremerhaven, 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
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>
</tbody>
</table>

Max 12

Time required: 12.
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 m p 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 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, now 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. Can ameliorate

- “Chunk” tasks to reduce overhead. May use large chunks initially, then decrease them near the end to fine-tune load balance.

- Use distributed queues, perhaps with
  - multiple manager/worker subteams, with some communication between managers
  - every worker is also a manager, keeping some tasks and sending extras to others. Many variations on deciding when/where to send work.
Recall that OpenMP has loop work-sharing constructs which require little programmer effort.

With the SCHEDULE option can specify

**STATIC:** simple, suitable if loop iterations take same amount of time and there are enough per processor. For scattered decomposition, specify chuck size.

**DYNAMIC:** a queue of work, each processor gets chunksize iterations when ready.

**GUIDED:** dynamic queue with chunks of exponentially decreasing size.
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.

Load-balancing needs to be approached as part of a systematic effort to improve performance.
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 needs to be approached as part of a systematic effort to improve performance.

Try simple approaches first.
Databases are an important commercial application of parallel computers, providing a base which helps keep some parallel computing companies viable.

Massive data collections important for Walmart, CERN Large Hadron Collider, Netflix, bioinformatics, . . .

Many of the ideas are used elsewhere, though sometimes obscured by different terminology. We’ll just briefly examine some aspects.
Characteristics

- Disk access and bandwidth dominates performance. Organizing the information to match the access patterns is often critical.

- Systems for scientific applications somewhat newer, complicated by factors such as being dispersed among sites, people trying to combine or mine information in new ways, billions of files (e.g., a constant stream of images), etc.

- Fault tolerance important
Common Architecture: Shared Some

Private

Shared

interconnection network

disk
RAM processor
disk
RAM processor
disk
RAM processor
disk
RAM processor
Data Mining

- Sifting for information in a torrent of data economically and scientifically important.

- AT&T, WalMart, Google, Amazon, ... have used for many years. Bioinformatics important new application area.

- Many commercial data mining tools, often parallelized.

- Warning: “data mining” means many different things to different people and applications.
Map-Reduce: New Form of Data Mining

- Variations used by Google, Yahoo, IBM, etc. Open source Hadoop: [http://hadoop.apache.org/core/](http://hadoop.apache.org/core/)

- Companies trying to get schools to teach this style of programming

- Basic database operations, extended to less organized, far larger, systems.

- Simple example: given records of (source page, link) for every company find # pages from outside the company that point to one of the company’s pages.
**Map:** determine if link record is from page outside a company into it. If so, generate new record (destination company, 1) embarrassingly parallel, vast number records, I/O bound

**Reduce:** combine records by company and sum the counts requires communication, but far fewer records

**Implementations:** significant emphasis on locality, efficiency, fault tolerance
Sample Map-Reduce Execution

Performance

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 operation per second” like Mflops, Gflops, Tflops or Petaflops.
Application-System Interplay

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

Application factors:
- Programming language
- 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
- Message passing library / OpenMP
- Communication pattern
- Task granularity
- Load balancing
Performance Gains: Hardware

Factor $\approx 10^4$ over the last 16 years
Performance Gains: Software

Gains expected from better algorithms, example:

Derived from Computational Methods (Linear Algebra)

Gains also expected from better load-balancing strategies, parallel I/O, etc.
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: general and floating point instructions, cache misses, branch miss prediction
- measures Mflop/s rate 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 3.7 released in 9/2009
- vendor independent hardware counter tool
- supports most current processors including the “Cell” processor
- user needs to instrument code ⇒ PAPI functions
- Fortran and C/C++ user interfaces
- easy-to-use and powerful high level API

Home page:

http://icl.cs.utk.edu/papi/index.html
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 opt**
  
  [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)
MPI and OpenMP Trace Tools

- Collect trace data at run time, display post-mortem
- Assess performance, bottlenecks and load-balancing problems in MPI & OpenMP 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://vampir.eu](http://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

Stout and Jablonowski – p. 247/284
here: uninstrumented version of the program

therefore: the routines of the user code can not be distinguished and are displayed as “Application”
Vampir Analysis – Zoom-in Timeline

Zoom-in: ⇒ Communication and synchronization
Global activity chart ⇒ Load-imbalance
Summary for the whole application: timing data
Vampir Analysis – MPI Summary

![VAMPiR Summary Chart](image)

- **Sum**: 18.509 s
- **MPI_Waitall**: 12.3 s
- **MPI_Isend**: 2.819 s
- **MPI_Allreduce**: 1.814 s
- **MPI_Irecv**: 0.819 s
- **MPI_Recv**: 0.336 s
- **MPI_Barrier**: 0.309 s
- **MPI_Send**: 96.605 ms
- **MPI_Reduce**: 15.568 ms
- **MPI_Comm_rank**: 0.304 ms
- **MPI_Comm_size**: 63.723 us
Public Domain Trace Tools

- Graphical displays of timelines, histograms, MPI overhead and more
- Instant zoom in/out, search/scan facility

TAU – Tuning and Analysis Utilities (version 2.18.3)
- Developed at the University of Oregon, mature
- Free, portable, open-source profiling/tracing facility ([http://www.cs.uoregon.edu/research/tau](http://www.cs.uoregon.edu/research/tau))
- Performance instrumentation, measurement and analysis toolkit for distributed and shared memory applications (includes MPI, OpenMP)
- Graphical displays for all or individual processes
- Manual or automatic source code instrumentation
Performance Analysis: Strategy

- **Hardware counters** provide information on Mflop/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

- **Increased parallel complexity** makes the debugging process 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:

- **TotalView** ([http://www.totalviewtech.com](http://www.totalviewtech.com))
- **allinea ddt** ([http://www.allinea.com](http://www.allinea.com))
Characteristics of Totalview

- Very powerful and mature debugger, current version 8.7
- Source-level, graphical debugger for C, C++, Fortran, High Performance Fortran (HPF) and assembler code
- Multiprocess (MPI) and multithread (OpenMP) codes
- Supports multi-platform applications
- Intuitive, easy-to-learn graphical interface
- Industry leader in MPI and OpenMP debugging
- Control functions to run, step, breakpoint, interrupt or restart a process
- 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 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
  - 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 Scientific Math Libraries

Parallel math libraries are available on most hardware platforms. Highly optimized and recommended.

**ScaLAPACK (Scalable LAPACK):**
- Public-domain, high-performance linear algebra routines for MPI applications
- Promotes modularity via interfaces to the libraries BLAS, BLACS and PBLAS

**NAG Parallel Libraries (commercial, often installed):**
- Mostly high speed linear algebra routines
- In addition: random number generation and quadrature routines

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

**ACTS toolkit** — Advanced CompuTational Software
(http://acts.nersc.gov):

- Public domain tools mostly developed at US labs
- Collection of tools that is interoperable, with API
- General solutions to complex programming needs
- Includes
  - **Numerical solvers**: PETSc, ScaLAPACK, Aztec, ...
  - **Structural frameworks**: Software that manages data & communication like Overture and Global Arrays
  - **Runtime & support tools**: CUMULUS, TAU

**Eclipse**: Parallel Tools Platform (PTP)

- open-source project: wide variety of parallel tools
In addition to programming, there are many issues concerning the use of parallel systems.

For example, they are often a centralized resource that must be shared, much like mainframes of olden days.

Your institution may decide to purchase a system, or buy time elsewhere.
Batch Queuing

A return to 60’s style computer usage.

- 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, you may be your own bottleneck. Might improve throughput by requesting fewer processors, and more time, per job (remember Amdahl’s Law).
**Access to Systems**

**Academics:** Can apply for free time at NSF supercomputing centers or perhaps your own university. For modest time the NSF process easy and quick, but thousands of hours requires more detailed application. Apply via Teragrid: [http://www.teragrid.org/](http://www.teragrid.org/)

Grants from other agencies usually include access to their large systems.

**Businesses:** Can purchase time from hardware vendors, sometimes from university centers.
Purchasing Systems

Buying systems very complicated. Typically total costs at least twice initial costs

Some questions:

- Can it run your major applications? May depend on ISVs.
- Will vendor be around in five years?
- Is there an upgrade path if you need to expand soon?
- Do you have sufficient power and air conditioning?
- Do you want more, or faster, processors, i.e., price-performance or performance?
Where are You on the Curve?

As a user or buyer, the type of processors you need to decrease running time:

![Graph showing the relationship between speedup, performance, and price/performance.](image)

Stout and Jablonowski – p. 266/284
Cluster Systems

For departmental-size machines, especially in academia:

- Many little things, hardware and software, go wrong or need upgrading — who will keep fixing this?
- Who does backups?

Maintenance time-consuming and harmful to career
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.
Power is Critical

- For given technology, typically $\text{power} \approx \text{speed}^2$
- Heat limits packing density
- Power In = Heat Out, so AC demands also increase
- Systems such as BlueGene make tradeoff: slower clock speed, less RAM/core, vastly more cores

*Tradeoff opposite programmer needs* — Amdahl’s law.
Power Trend

Scaling clock speed (business as usual) will not work

Source: Patrick Gelsinger, Intel®
Chip Density Still Increasing

- Hardware designers running out of old tricks, so just replicate processors on chips — multi-core, many-core.
- While potential chip performance continues to increase, number of I/O wires/chip doesn’t match number cores, more stress on cache locality.
- GPUs (and IBM Cell) have large number of simple processors, very high FLOPs, again need locality for efficient vector-like operations.
- IBM Roadrunner has 129,600 cores
- Exascale systems will have 100,000,000
Chip Trends

Sources: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Good Optimization still Bleeding Edge

- Economics says use commodity parts. Unfortunately
  - GPUs difficult to program
  - Many-cores will be quite complex to use efficiently

- Some new languages focus on parallelism: IBM’s X10 (support integrated into Eclipse), Cray’s Chapel, but historically new languages have low acceptance rate

- Need better compilers to exploit parallelism (e.g., much smarter OpenMP compilers)

- Mixed mode programming, MPI plus (OpenMP, CUDA, OpenCL), needed for highest performance
More Trends

- Roadrunner grabs the headlines, but clusters and SMPs most important economically.

- Increasing use of commercial parallelized software.

- $/flop continues to plummet: for 1 GFLOP, in 1961 the cost was $1,100,000,000,000. Now it is about $0.13. [http://en.wikipedia.org/wiki/FLOPS](http://en.wikipedia.org/wiki/FLOPS).

- FLOPS/Watt a new measure of interest. Green500 list, best computer: 536 MFLOPS/W, worst: 13 MFLOPS/W. GPGPUs at the high end of this range. [http://www.green500.org/](http://www.green500.org/)

- Some parallel computing companies will fail.
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.
If It Isn’t Working Well . . .

- The original program probably wasn’t written with parallelism in mind.

- See if there is a more parallelizable approach.

- Sometimes parallelizable approaches aren’t the most efficient ones available for serial computers, but that is OK if you are going to use many processors.
If It Isn’t Working Well . . .

- The original program probably wasn’t written with parallelism in mind

- See if there is a more parallelizable approach

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

Remember Amdahl’s Law:

Efficient massive parallelism is difficult.
Finally ...
Finally ...

Make sure your 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

- Green500: [http://www.green500.org/](http://www.green500.org/)
- Hilbert space-filling curve: see the routines available in Zoltan (listed below).
References continued

  Free, portable versions at:

- 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)


References continued

- Top500. Website with extensive collection of references: [http://www.Top500.org](http://www.Top500.org)
- UPC (Unified Parallel C): [http://upc.gwu.edu](http://upc.gwu.edu)