Three basic multiprocessing issues

1. *Partitioning.* The sequential program must be *partitioned* into subprogram units or tasks. This is done either by the programmer or by the compiler.

2. *Scheduling.* Associated with the tasks in a program are requirements on when the tasks must execute.
   - Some tasks may be executed at any time.
   - Some tasks may not be initiated before certain other tasks have completed.

   At run time, the scheduler attempts to arrange the tasks in such a way as to minimize the overall program execution time.

3. *Synchronization and communication.* In order for the program to be *correct*, different tasks have to be synchronized with each other. The architecture has to provide primitives for synchronization.

   In order to be efficient, the tasks must communicate rapidly.

   An important communication issue is memory *coherence*: If separate copies of the same data are held in different memories, a change to one must be “immediately” reflected in the others.

Exactly how efficient the communication must be depends on the *grain size* of the partitioning.

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Program construct</th>
<th>Typical # instrs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine grain</td>
<td>Basic block</td>
<td>5–10</td>
</tr>
<tr>
<td></td>
<td>Procedure</td>
<td>20–100</td>
</tr>
<tr>
<td>Medium grain</td>
<td>Small task</td>
<td>100–1,000</td>
</tr>
<tr>
<td></td>
<td>Intermediate task</td>
<td>1,000–100,000</td>
</tr>
<tr>
<td>Coarse grain</td>
<td>Large task</td>
<td>&gt; 100,000</td>
</tr>
</tbody>
</table>
A parallel program runs most efficiently if the grain size is neither too large nor too small.

What are some disadvantages of having a too-large grain size?

• 
  
• 

What would be some disadvantages of making the grain size too small?

• 
  
• 

The highest speedup is achieved with grain size somewhere in between.

The graph illustrates the relationship between speedup and grain size, showing that:

- Limited parallelism and load imbalance occur at the coarse end.
- High overhead is present at the fine end.

Task overheads depend on the cache organization.

- A large cache means a _____ task overhead.
- Shared caches tend to _________ the overhead.

A task returning to a processor that still has many of its instructions and data in its cache will face a lower overhead than if it started on a different processor.

A program is broken into tasks in one of the following ways:
• Explicit statement of concurrency in the higher-level language, such as in CSP or occam. These allow programmers to divide their programs into tasks and specify the communications between them.

• The use of hints in the source code, which the compiler may choose to use or ignore.

• Parallelism is detected by a sophisticated compiler, such as Parafrase. These use techniques like data-dependency graphs to transform serial code into parallel tasks.

Sometimes “too much” parallelism is found in a program; then tasks are clustered for efficient execution.

• This may be done when there aren’t enough processors to execute all the tasks in parallel.

• It may be profitable to group the tasks if their working sets intersect heavily.

**Multiprocessors vs. networks**

In a multiprocessor, the different processors (nodes) are under the control of a single operating system.

In a network (e.g., local-area network) each node is autonomous and can run without the assistance of a central operating system (though it may not have access to all the peripherals).

In general, a network is physically distributed while a multiprocessor is centralized.

In a multiprocessor, all processors do not have to be up in order for the system to run. But not every processor can do useful work by itself (e.g., it may not be attached to any peripherals at all).

By contrast, network nodes often communicate by transferring files (although some local-area networks now support paging).
Tightly coupled vs. loosely coupled multiprocessors

The next step up from a network is a multiprocessor that has no shared memory. Such a multiprocessor is called *loosely coupled*.

Communication is by means of inter-processor *messages*. A loosely coupled multiprocessor is often called a “message-passing” or “distributed-memory” multiprocessor.

What is Gustafson’s term for this?

A multiprocessor is called *tightly coupled* if it has a shared memory. The communication bandwidth is on the order of the memory bandwidth. A tightly coupled multiprocessor is often called a “shared-memory” multiprocessor.

What is Gustafson’s term for this?

Tightly coupled processors support a _______ grain size than loosely coupled processors.

Tightly coupled multiprocessors share memory in several different ways. Among these are—

- Shared data cache/shared memory.
- Separate data cache but shared bus to memory.
- Separate data caches with separate buses to shared memory.
- Separate processors and separate memory modules interconnected via a multistage switch.
- A separate multilevel cache *instead* of a shared memory.

There is a middle ground between tightly coupled and loosely coupled multiprocessors, machines with—

- a private memory for each processor, where
- processors are capable of accessing each others’ memories with a certain time penalty.
Sometimes these machines are called non-uniform memory access (NUMA) machines. The BBN Butterfly™ is an early example. A more recent example is the Silicon Graphics Origin architecture.

In any tightly coupled multiprocessor, the processors are all connected to the memory and peripherals via some sort of interconnection structure, or interconnect. This looks like this:

This is sometimes implemented by providing a “back door” to allow the processor to bypass the interconnection structure for accesses to its private memory.

It can be done for any multistage interconnection network, as well as the crossbar shown at the right.

Another approach is to build a system in clusters.
Communication architectures

Each type of parallel architecture has developed its own programming model.

It is useful to visualize a parallel architecture in terms of layers:

<table>
<thead>
<tr>
<th>CAD</th>
<th>Database</th>
<th>Scientific modeling</th>
<th>Parallel applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprogramming</td>
<td>Shared address</td>
<td>Message passing</td>
<td>Data parallel</td>
</tr>
<tr>
<td>Compilation or library</td>
<td>Operating systems</td>
<td>Communication abstraction</td>
<td></td>
</tr>
<tr>
<td>Communication hardware</td>
<td>Physical communication medium</td>
<td>User/system boundary support</td>
<td></td>
</tr>
</tbody>
</table>

Three parallel programming models stand out.
• **Shared-address** programming is like using a “bulletin board” where you can communicate with colleagues.

• **Message-passing** is like communicating via e-mail or telephone calls. There is a well defined event when a message is sent or received.

• **Data-parallel** programming is a “regimented” form of cooperation. Many processors perform an action separately on different sets of data, then exchange information globally before continuing en masse.

User-level communication primitives are provided to realize the programming model

• There is a mapping between language primitives of the programming model and these primitives

These primitives are supported directly by hardware, or via OS, or via user software.

In the early days, the kind of programming model that could be used was closely tied to the architecture.

Today—

• Compilers and software play important roles as bridges
• Technology trends exert a strong influence

The result is convergence in organizational structure, and relatively simple, general-purpose communication primitives.

**A shared address space**

In the shared address-space model, processes can access the same memory locations.

Communication occurs implicitly as result of loads and stores

This is convenient.

• Wide range of granularities supported.
• Similar programming model to time-sharing on uniprocessors, except that processes run on different processors

Good throughput on multiprogrammed workloads.

Naturally provided on wide range of platforms.

• History dates at least to precursors of mainframes in early 60s
• Wide range of scale: few to hundreds of processors

This is popularly known as the shared memory model.

But this term is ambiguous. Why?

*The shared address-space model*

A process is a virtual address space plus

Portions of the address spaces of tasks are shared.

![Diagram of shared address-space model](image-url)
What does the private region of the virtual address space usually contain?

Conventional memory operations can be used for communication.

Special atomic operations are used for synchronization.

*The interconnection structure*

The interconnect in a shared-memory multiprocessor can take several forms.

It may be a *crossbar switch*.

Each processor has a direct connection to each memory and I/O controller.

Bandwidth scales with the number of processors.

Unfortunately, cost scales with

CS&G call this the “mainframe approach.”

At the other end of the spectrum is a *shared-bus* architecture.

All processors, memories, and I/O controllers are connected to the bus.

Such a multiprocessor is called a symmetric multiprocessor (SMP).

* Latency larger than for uniprocessor
• Low incremental cost
• Bus is bandwidth bottleneck

A compromise between these two organizations is a multistage interconnection network.

The processors are on one side, and the memories and controllers are on the other.

Each memory reference has to traverse the stages of the network.

Why is this called a compromise between the other two strategies?

For small configurations, however, a shared bus is quite viable.

Current shared-address architectures

Example: Intel Pentium Pro four-processor “quad pack.”

Each of the four processors contains—
• a Pentium-pro processor,
• translation lookaside buffer,
• first-level caches,
• a 256-KB second-level cache,
• an interrupt controller, and
• a bus interface.

All of these are on a single chip connecting directly to a 64-bit memory bus.

*Example*: Sun Enterprise.

This is based on a 256-bit memory bus, delivering 2.5 GB/s of memory bandwidth.

There may be up to 16 cards, where each card is either a CPU/memory card, or complete I/O system.

CPU/memory card contains two UltraSparc processor modules, each with

• 16-KB level-1 and 512-KB level-2 caches,
• two 512-bit wide memory banks, and
• an internal switch.

The I/O card contains

• three SBus slots for I/O extensions,
• a SCSI connector,
• a 100 bT Ethernet port, and
• two FiberChannel interfaces.
Although memory is packaged with the processors, it is always accessed over the common bus, making this a uniform memory-access multiprocessor.

A full configuration would be 24 processors and 6 I/O cards.

*The Cray T3E:* For larger numbers of processors, a NUMA design is essential. It is designed to scale up to 1024 processors, with 480 MB/s communication links.

- Processors are DEC Alphas.
- Topology is a 3-dimensional cube, with each node connected to its six neighbors.

Any processor can access any memory location.

A short series of instructions is used to establish addressability to remote memory, which can then be accessed via conventional loads and stores.

Remote data is not cached—there is no way to keep it coherent.