Francesco Quaglia

Architetture Avanzate di Elaborazione
Corso di Laurea in Ingegneria Informatica
Facolta’ di Ingegneria, Universita’ “La Sapienza”

Contents:
1. Parallel programming models
2. Design of parallel algorithms
3. Quantitative basis for design and evaluation
4. Parallel programming environments: MPI
Disclosure
Most of the contents of this presentation have been taken from the on-line course:

“Designing and Building Parallel Programs”
by Ian Foster
(http://www-unix.mcs.anl.gov/dbpp/)
Parallel programming models
Abstractions

• A parallel programming model is a set of reference abstractions for the design of parallel algorithms

• These abstractions need to be simple enough to work with and need to match the general multicomputer architectural model (e.g. SMP vs cluster)

• They also need to provide adequate supports for explicit discussion about
  - concurrency
  - locality

• This is required to facilitate the development of scalable and modular programs
The “tasks and channels” model

1. A parallel computation consists of one or more tasks executing concurrently, whose number can vary during program execution.

2. A task encapsulates a sequential program and local memory (it is a virtual Von-Neumann machine).

3. A set of *inports* and *outports* define its interface to its environment.

4. A task can perform four basic actions in addition to reading and writing its local memory:
   - send messages on its outports
   - receive messages on its inports
   - create new tasks
   - terminate
Send a message:

Receive a message:

Create tasks:

Terminate:
4. Outport/inport pairs can be connected by message queues called *channels*

5. Channels can be created and deleted, and references to channels (ports) can be included in messages, so connectivity can vary dynamically

5. Tasks can be mapped to physical processors in various ways; the mapping employed does not affect the semantics of a program. In particular, multiple tasks can be mapped to a single processor
Peculiarities

• The task abstraction provides a mechanism for talking about both concurrency and locality (since we can distinguish between local and remote data)

• The channel abstraction provides a mechanism for indicating that computation in one task requires data in another task in order to proceed. This is termed a *data dependency*
“Tasks and channels” model properties

1. **Performance**: If two tasks that share a channel are mapped to different processors, the channel connection is implemented as interprocessor communication; if they are mapped to the same processor, some more efficient mechanism can be used.

2. **Mapping Independence**: Because tasks interact using the same mechanism (channels) regardless of task location, the result computed by a program does not depend on where tasks execute. Hence, algorithms can be designed and implemented without concern for the number of processors on which they will execute.
4. **Modularity**: Different tasks can be developed separately, as independent modules. Hence, module implementations can be changed without modifying other components, and the properties of a program can be determined from the specifications for its modules and the code that plugs these modules together.

5. **Determinism**: An algorithm or program is deterministic if execution with a particular input always yields the same output. We can get determinism by having, e.g., FIFO channels, a single sender/receiver task per channel and blocking receive operations.
Other parallel programming models

- **Data Parallelism:**
  - a data-parallel program consists of a sequence of identical operations applied to multiple elements of a data structure
  - As each operation on each data element can be thought of as an independent task, the natural granularity of a data-parallel computation is small, and the concept of “locality” does not arise naturally
  - Data-parallel compilers often require the programmer to provide information about how data are to be distributed over processors. The compiler can then translate the data-parallel program into an SPMD formulation, thereby generating communication code automatically
  - Example supports: *High Performance Fortran*
• **Shared Memory:**

  - In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.

  - Various mechanisms such as locks and semaphores may be used to control access to the shared memory.

  - An advantage of this model from the programmer's point of view is that the notion of data “ownership” is lacking, and hence there is no need to specify explicitly the communication of data.

  - However, understanding and managing locality becomes more difficult (data access conflicts).

  - It can also be more difficult (and less effective) to write deterministic programs.
• **Message passing:**
  
  ➢ Message passing is probably the most widely used parallel programming model today
  
  ➢ Message passing is really just a minor variation on the task/channel model, differing only in the mechanism used for data transfer (messages are destined to tasks instead of being sent over channels)
  
  ➢ Example supports: *Message Passing Interface (MPI)*
  
  ➢ The message-passing model does not preclude the dynamic creation of tasks, the execution of multiple tasks per processor, or the execution of different programs by different tasks. However, in practice some message-passing systems create a fixed number of identical tasks (SPMD like) at program startup and do not allow tasks to be created or destroyed during program execution
Design of parallel algorithms
Methodical design

• Most programming problems have several parallel solutions. Therefore, the design methodology is intended to foster an exploratory approach to design in which machine-independent issues such as concurrency are considered early and machine-specific aspects of design are delayed until late in the design process.

• One classical methodology (PCAM) structures the design process as four distinct stages:
  - partitioning & communication (with focus on concurrency and scalability)
  - agglomeration & mapping (with focus on locality and other performance related issues)
PCAM steps

1. **Partitioning.** The attention is focused on recognizing opportunities for parallel execution by decomposing the computation that is to be performed and the data operated on by this computation into “small” tasks. Practical issues such as the number of processors in the target computer are ignored.

2. **Communication.** The communication required to coordinate task execution (data dependency) is determined and the proper communication structures and algorithms are identified.
3. *Agglomeration*. The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

4. *Mapping*. Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.
An example
Partitioning

• The focus is on defining a large number of small tasks in order to yield what is termed a *fine-grained* decomposition of a problem

• In later design stages, evaluation of communication requirements, the target architecture, or software engineering issues may lead us to forego opportunities for parallel execution identified at this stage

• A good partition divides into small pieces both the *computation* associated with a problem and the *data* on which this computation operates
Partitioning approaches

• Programmers most commonly first focus on the data associated with a problem, then determine an appropriate partition for the data, and finally work out how to associate computation with data. This partitioning technique is termed *domain decomposition*

• The alternative approach, termed *functional decomposition*, first decomposes the computation to be performed and then deals with the data

• These are complementary techniques which may be applied to different components of a single problem or even applied to the same problem to obtain alternative parallel algorithms
Issues with domain decomposition

- If possible, the data are divided into small pieces of approximately equal size

- This partitioning yields a number of tasks, each comprising some data and a set of operations on that data

- The main focus can be on either on the largest data structure (effectiveness of virtual memory and I/O) or the data structure accessed more frequently (effectiveness of the caching hierarchy)

- An operation may require data from several tasks. In this case, communication is required to move data between tasks
Issues with functional decomposition

- Main activities to be performed by the computation are identified, and related tasks are generated.

- The related data requirements may be disjoint, in which case the partition is complete. Alternatively, they may overlap significantly, in which case considerable communication will be required.

- Functional decomposition also has an important role to play as a program structuring technique. A functional decomposition that partitions not only the computation that is to be performed but also the code that performs that computation is likely to reduce the complexity of the overall design.
Partitioning check list

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.

2. Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.

3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.

4. Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. Otherwise, your parallel algorithm may not be able to solve larger problems when more processors are available.

5. Have you identified several alternative partitions? You can maximize flexibility in subsequent design stages by considering alternatives now.
Communication

• The focus is on the identification of possible data dependency among different tasks and on the identification of all required channels

• Communication patterns can then be categorized along three loosely orthogonal axes:
  - local/global (depending on the amount of tasks each task communicates with)
  - structured/unstructured (depending on whether a task and its neighbors form a regular structure, such as a tree or grid)
  - static/dynamic (depending on whether the identity of communication partners changes over time)
Communication checklist

1. Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggest a nonscalable construct. Revisit your design to see whether communication operations can be distributed more equitably. For example, if a frequently accessed data structure is encapsulated in a single task, consider distributing or replicating this data structure.

2. Does each task communicate only with a small number of neighbors? If each task must communicate with many other tasks, evaluate the possibility of formulating this global communication in terms of a local communication structure.

3. Are communication operations able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable.

4. Is the computation associated with different tasks able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Consider whether you can reorder communication and computation operations.
**Agglomeration**

- After partitioning and communication, the resulting algorithm is still abstract in the sense that it is not specialized for efficient execution on any particular parallel computer.

- In fact, it may be highly inefficient if, for example, it creates many more tasks than there are processors on the target computer and this computer is not designed for efficient execution of small tasks and/or they cannot execute concurrently due to, e.g., data dependency.

- *Agglomeration* revisits decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer.

- It considers whether it is useful to combine, or *agglomerate*, tasks identified by the partitioning phase, so as to provide a smaller number of tasks, each of greater size.
Agglomeration effects

• It tends to increase the task granularity hence tending to reduce communication costs via reduction of the number of messages

• This reduction is mastered by the *surface-to-volume effect*:
  
  ➢ the communication requirements of a task are proportional to the surface of the subdomain on which it operates, while the computation requirements are proportional to the subdomain's volume

  ➢ e.g. in a two-dimensional problem, the “surface” scales with the problem size while the “volume” scales as the problem size squared (i.e. the amount of communication performed for a unit of computation - *communication/computation ratio* - decreases as task size increases)
A consequence of surface-to-volume effects is that higher-dimensional decompositions are typically the most efficient, other things being equal, because they reduce the surface area (communication) required for a given volume (computation).

Hence, from the viewpoint of efficiency it is usually best to increase granularity by agglomerating tasks in all dimensions rather than reducing the dimension of the decomposition.
One example

Communication is reduced from 8x8x4 to 4x4
.... secondary “message aggregation” effects

• When task granularity is increased via agglomeration there is the possibility that each message size is also increased

• The real effect is however in favor of communication cost reduction due to the fact that common systems have a communication cost composed by
  ➢ A fixed setup cost and
  ➢ A per byte data transfer cost

• The setup cost may be up to an order of magnitude greater than the per byte data transfer cost, which yields communication cost reduction aggregating several data units into a single message

• We will come back on so called primary message aggregation effects later
Agglomeration checklist

1. Has agglomeration reduced communication costs by increasing locality? If not, examine your algorithm to determine whether this could be achieved using an alternative agglomeration strategy.

2. Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created, the more important it is that they have similar costs. If we have created just one task per processor, then these tasks should have nearly identical costs.

3. Does the number of tasks still scale with problem size? If not, then your algorithm is no longer able to solve larger problems on larger parallel computers.

4. If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers? An algorithm with insufficient concurrency may still be the most efficient, if other algorithms have excessive communication costs.
Mapping

• The mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling (e.g. load sharing) but, unfortunately, general-purpose mapping mechanisms have yet to be developed for scalable parallel computers

• The goal in developing mapping algorithms is normally to minimize total execution time via two strategies:
  ➢ Placing tasks that are able to execute concurrently on different processors, so as to enhance concurrency.
  ➢ Placing tasks that communicate frequently on the same processor, so as to increase locality.

• These two strategies will sometimes conflict and resource limitations may restrict the number of tasks that can be placed on a processor

• The mapping problem is known to be \textit{NP–complete}, however, there are strategies and heuristics effective for several classes of problems
Mapping approaches

For domain decomposition:

- In case of a fixed number of equal-sized tasks and structured local and global communication an efficient mapping is straightforward; we may also choose to agglomerate tasks mapped to the same processor, if this has not already been done, to yield a total of $P$ coarse-grained tasks, one per processor.

- In more complex domain decomposition-based algorithms with variable amounts of work per task and/or unstructured communication patterns, we may employ load balancing.

- The most complex problems are those in which either the number of tasks or the amount of computation or communication per task changes dynamically during program execution since we need to use a dynamic load-balancing (local vs global knowledge based).
For functional decomposition:

- We may get computations consisting of many (short-lived) tasks that coordinate with other tasks only at the start and end of execution.

- In this case, we can use task-scheduling algorithms, which allocate tasks to processors that are idle or that are likely to become idle.
(Dynamic) Load balancing algorithms

- Recursive bisection
  - Recursive coordinate bisection
  - Unbalanced recursive bisection
  - Recursive graph bisection

- Local algorithms

- Probabilistic methods

- Cyclic mapping
Recursive bisection

- Recursive bisection techniques are used to partition a domain (e.g., a finite element grid) into subdomains of approximately equal computational cost while attempting to minimize communication costs, i.e., the number of channels crossing task boundaries.

- A divide-and-conquer approach is taken.

- The domain is first cut in one dimension to yield two subdomains. Cuts are then made recursively in the new subdomains until we have as many subdomains as we require tasks.
Recursive coordinate bisection

- It is the most straightforward of the recursive bisection techniques
- It is normally applied to irregular grids that have a mostly local communication structure
- It makes cuts based on the physical coordinates of grid points in the domain, at each step subdividing along the longer dimension
- This approach has the advantages of being simple and inexpensive. It also does a good job of partitioning computation
- A disadvantage is that it does not optimize communication performance (it can generate long, skinny subdomains, which if an algorithm has significant local communication will result in more messages than will a decomposition that generates square subdomains)
Unbalanced recursive bisection

• It is a variant of recursive bisection

• It attempts to reduce communication costs by forming subgrids that have better aspect ratios (i.e. communication/computation ratios)

• It considers the \( P-1 \) partitions obtained by forming unbalanced sub-grids with \( I/P \) and \( (P-1)/P \) of the load, with \( 2/P \) and \( (P-2)/P \) of the load, and so on, and chooses the partition that minimizes partition aspect ratio

• This method increases the cost of computing the partition but can reduce communication costs
Recursive graph bisection

- It can be useful in the case of more complex unstructured grids

- This technique uses connectivity information to reduce the number of grid edges crossing subdomain boundaries, and hence to reduce communication requirements

- A grid is treated as a graph with N vertices (grid points)

- The algorithm first identifies the two extremities of the graph, that is, the two vertices that are the most separated in terms of graph distance. (The graph distance between two vertices is the smallest number of edges that must be traversed to go between them)

- Each vertex is then assigned to the subdomain corresponding to the closer extremity
Load balancing vs virtual processors

• In some cases, the available processors may be heterogeneous, with the meaning that the units of work to be assigned to each processor might be different to achieve well load distribution.

• More important, in non-dedicated environments, the fraction of computational power assigned to the parallel application may vary due to load sources related to external applications.

• One way to cope with this is the “virtual processor” concept, which assigns to a processor $P_i$ a function $X_i(t)$ expressing the relative computational power available for the parallel application at time $t$.

• Variation of $X_i(t)$ can be used as a trigger for (re)executing the load balancing algorithm.
Decision policies for load balancing triggering

\[ \Delta_i(t) = |\chi_i(t) - \chi_i(t_l)| \]

Average system imbalance

\[ \Delta(t) = \frac{1}{p} \sum_{i=1}^{p} \Delta_i(t) > \vartheta_a \]

Maximum system imbalance

\[ \Delta_i(t) = |\chi_i(t) - \chi_i(t_l)| > \vartheta_b \]

Number of imbalanced nodes

\[ \Delta_i(t) = \begin{cases} 1 & \text{if } |\chi_i(t) - \chi_i(t_l)| > \lambda \\ 0 & \text{otherwise} \end{cases} \]
Overhead for load balancing

• In depends on the complexity of the selected algorithm plus
  ➢ The complexity of the monitoring system providing imbalance information
  ➢ The type of collection (synchronous vs asynchronous) for imbalance information in terms of
    ✓ Synchronization barriers (and related algorithms)
    ✓ Messages for triggering synchronization barriers
• Threshold values in the decision policies play therefore a central rule
Local algorithms

- Local load-balancing algorithms compensate for changes in computational load using only information obtained from a (small) number of neighboring processors.

- For example, processors may be organized in a logical mesh: each processor compares its computational load with that of its neighbors in the mesh and transfers computation if the difference in load exceeds some threshold.

- These algorithms may result extremely effective especially when the data domain size (and thus load) constantly changes during computation.

- In such a case they might even allow ideal partitioning with no explicit partitioning step performed at all.
• Care must be taken in designing a load balancing algorithm not prone to the risk of overloads due to bad “instantaneous decisions”

• Information about load on different processors can be exchanged in such a way to obtain
  ➢ Perfect load information (typically inviable)
  ➢ Approximate load information (e.g. periodically)

• Approximate load information can be achieved also with so called message piggybacking techniques
  ➢ Load information is associated with messages carrying application data
Probabilistic methods

• Tasks are allocated to randomly selected processors
• If the number of tasks is large, we can expect that each processor will be allocated about the same amount of computation
• Advantages of this strategy are its low cost and scalability
• Disadvantages are that off-processor communication is required for virtually every task and that acceptable load distribution is achieved only if there are many more tasks than there are processors
• The strategy tends to be most effective when there is relatively little communication between tasks and/or little locality in communication patterns
Cyclic mapping

- Each of $P$ processors is allocated every $P$-th task according to some enumeration of the tasks.

- This technique is a form of probabilistic mapping. The goal is that, on average, each processor will be allocated about the same computational load.

- The benefits of improved load balance may need to be weighed against increased communication costs due to reduced locality.

- Block cyclic distributions are also possible, in which blocks of tasks are allocated to processors.
Task scheduling algorithms

• A centralized or distributed task pool is maintained, into which new tasks are placed and from which tasks are taken for allocation to processors

• Tasks can be reinterpreted as data structures representing “problems” to be solved by a set of worker tasks, typically one per processor

• The strategy to allocate problems to workers represents a compromise between the conflicting requirements for independent operation (to reduce communication costs) and global knowledge of computation state (to improve load balance)

• Approaches:
  ➢ manager/worker
  ➢ hierarchical manager/worker
  ➢ decentralized approaches
Manager/worker

- A central manager task is given responsibility for problem allocation.
- Each worker repeatedly requests and executes a problem from the manager.
- Workers can also send new tasks to the manager for allocation to other workers. The efficiency of this strategy depends on the number of workers and the relative costs of obtaining and executing problems.
- Typically effective for moderate numbers of processors.
- The effectiveness can be improved by pre-fetching problems so as to overlap computation and communication, and by caching problems in workers, so that workers communicate with the manager only when no problems are available locally.
Hierarchical manager/worker

• It is a variant of the manager/worker scheme
• It divides workers into disjoint sets, each with a sub-manager
• Workers request tasks from sub-managers,
• Sub-manager communicate periodically with the manager and with other sub-managers to balance load between the sets of processors for which they are responsible
Decentralized schemes

- In completely decentralized schemes, there is no central manager. Instead, a separate task pool is maintained on each processor, and idle workers request problems from other processors.

- In effect, the task pool becomes a distributed data structure that is accessed by the different tasks in an asynchronous fashion.

- A variety of access policies can be defined:
  - a worker may request work from a small number of predefined “neighbors” or
  - may select other processors at random.

- In a hybrid centralized/distributed scheme, requests are sent to a central manager allocating them to workers in a round-robin fashion (while this manager will certainly be a bottleneck on large systems, it will typically be accessed less frequently than the manager in a manager/worker and hence increases scalability).
Termination detection

- Task-scheduling algorithms require a mechanism for determining when a search is complete; otherwise, idle workers will never stop requesting work from other workers.
- This *termination detection* operation is straightforward in centralized schemes, because the manager can easily determine when all workers are idle.
- It is more difficult in decentralized algorithms, because not only is there no central record of which workers are idle, but also messages in transit may be carrying tasks even when all workers appear to be idle.
The graph reduction case

- Graph reduction means computing a final graph by starting from an initial one and rewriting the graph on the basis of, e.g., topological features.

- In graph reduction problems the size of the data set (e.g. number of nodes or edges) on which the program operates may change significantly.

- Mapping, typically in the form of dynamic task scheduling, becomes then a core step for achieving effective parallel execution.

- Local algorithms are typically preferred whenever reduction steps exhibit relatively fine granularity.
Data set variation example

$\text{UPM} = \# \text{Un-Processed Messages (i.e. pending edges)}$
A local task scheduling algorithm

• Each task (Pi) keeps track of the values of upm related to itself and to the other processes into a vector UPMi

• UPMi[i] records the current value of the number of unprocessed application messages of Pi

• UPMi[j] records the value of the number of unprocessed application messages of Pj known by Pi

• Whenever Pi sends a physical message M to Pj, the value of UPMi[i] is piggy-backed on the message (denoted M.UPM)

• Whenever a physical message M sent by Pj to Pi is received from Pi, then UPMi[j] is updated from M.UPM
• Load distribution rule:

  ➢ Pi keeps a counter $RR_i$ initialized to zero which is updated (module n) each time a new node is produced by Pi.

  ➢ The current value of $RR_i$ is the identifier of the process which should host the new node according to the roundrobin policy.

  ➢ Pi actually select that process as destination if $UPM_i[RR_i] < UPM_i[i] $; otherwise Pi selects itself as destination for the new node.

• Would the selection of the least loaded process be a better solution for this load balancing problem?
Termination detection

program $P_i$;
1    initialize();
2   while not end_computation do
3      (collect all incoming messages and store them in $\text{incoming}_i$)
4       while not empty($\text{incoming}_i$) do
5          (extract a message $m$ from $\text{incoming}_i$);
6          if $m\.target \in \text{nodes}_i$  'node already in the local list'
7             then
8               for each edge $e \in \text{nodes}_i(m\.target).\text{combusted}$ do
9                   if $\text{Side}(e_m) \neq \text{Side}(e)$
10                      then
11                          (compose $e_m$ with $e$);
12                          (select the destination process $P_j$ for hosting the node possibly originated by the composition);
13                          (send the edges produced by the composition to $P_k$ and $P_h$
14                             hosting $m\.source$ and $e\.source$ respectively)
15               endfor
16           else (add $m\.target$ to $\text{nodes}_i$);  'delayed creation'
17               (add $e_m$ to $\text{nodes}_i(m\.target).\text{combusted}.\text{Side}(e_m)$)
18      endwhile;
19    endwhile
20 (end_computation = check_termination());
21 endwhile
• Each message can also be seen as a problem
• The computation is over when all problems have been solved
• Termination detection scheme:
  ➢ The master sends terminate messages to the slaves
  ➢ The slaves send status messages to the master carrying information about sent/received and solved problems
• Computation is over when the master has detected a perfect matching between injected and solved problems
Mapping design checklist

1. If considering an SPMD design for a complex problem, have you also considered an algorithm based on dynamic task creation and deletion? The latter approach can yield a simpler algorithm, however, performance can be problematic.

2. If considering a design based on dynamic task creation and deletion, have you also considered an SPMD algorithm? An SPMD algorithm provides greater control over the scheduling of communication and computation, but can be more complex.

3. If using a centralized load-balancing scheme, have you verified that the manager will not become a bottleneck? You may be able to reduce communication costs in these schemes by passing pointers to tasks, rather than tasks, to the manager.
Quantitative basis for design and evaluation
“Performance” of a parallel program

- The “performance” of a parallel program is a complex and multifaceted issue
- In addition to the execution time and scalability, one should also consider
  - The mechanisms by which data are generated, stored, transmitted over networks, moved to and from disk
  - Costs incurred at different phases of the software life cycle, including design, implementation, and maintenance
  - Hardware requirements and portability
- Nevertheless, we focus on execution time and parallel scalability because they are frequently among the more problematic aspects of parallel program design
Developing evaluation models: why and how

- Models are the basis for the comparison among different parallel programming choices
- They also allow to optimize the effectiveness of each single solution by both static and dynamic optimized parameter selection
- When dealing with parallel programming, classical “asymptotic cost” models reveal not adequate since unconsidered costs are no more representable via constants
- An adequate model should express execution time $T$ as a function of
  - problem size $N$
  - number of processors $P$
  - number of tasks $U$
  - other algorithm and hardware characteristics
Expressing the execution time

• The execution time of a parallel program is defined as the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution.

• In case of time sharing with other applications on the same hardware, such a definition should be modified in order to capture contention effects of resources.

• During execution, a processor is computing, communicating, or idling.

• Total execution time \( T \) can be defined:
  - as the sum of computation, communication, and idle times on an arbitrary processor \( j \), \( T = T_{\text{comp}}^j + T_{\text{comm}}^j + T_{\text{idle}}^j \),
  - or as the sum of these times over all processors divided by the number of processors \( P \), \( T = \frac{1}{P} (T_{\text{comp}} + T_{\text{comm}} + T_{\text{idle}}) 
    = \frac{1}{P} \left( \sum_{i=0}^{P-1} T_{\text{comp}}^i + \sum_{i=0}^{P-1} T_{\text{comm}}^i + \sum_{i=0}^{P-1} T_{\text{idle}}^i \right) \).
An example

- = Computation
- = Communication
- = Idle
Tricks

• The expressions of involved terms should be as simple as possible, while providing acceptable accuracy.

• The following approaches can be used to reduce model complexity:
  
  - Low-level hardware details such as memory hierarchies and the topology of the interconnection network are introduced only if there is evidence to suggest that they are important (e.g. the expression holds only for fully associative cache).
  
  - Scale analysis can be used to identify insignificant effects that can be ignored in the analysis. For example, if an algorithm consists of an initialization step followed by several thousand iterations of a computation step, then unless initialization is very expensive we can consider only the computation step in our analysis.
  
  - Empirical studies can be used as the basis to calibrate simple models rather than developing more complex models from first principles.
Computation time: basics

- Computation time will normally depend on some measure of problem size, whether that size is represented by a single parameter $N$ or by a set of parameters.
- If the parallel algorithm replicates computation, then computation time will also depend on the number of tasks or processors.
- In a heterogeneous parallel computer (such as a workstation network), computation time can vary according to the processor on which computation is performed.
- Computation time will also depend on characteristics of processors and their memory systems. For example, scaling problem size or number of processors can change cache performance or the effectiveness of processor pipelining.
- As a consequence, one cannot automatically assume that total computation time will stay constant as the number of processors changes.
Communication time: basics

• Interprocessor and intraprocessor communication costs can be typically assumed as comparable

• Perhaps surprisingly, this assumption is not unreasonable in many multicomputers, unless
  ➢ intraprocessor communication is highly optimized
  ➢ network is under heavy access conflict (e.g. there is significant bandwidth contention)

• This is because the cost of the memory-to-memory copies and context switches performed in a typical implementation of intraprocessor communication is often comparable to the cost of an interprocessor communication
• the cost of sending a message between two tasks located on different processors can be represented by two parameters:
  ➢ the message startup time, which is the time required to initiate the communication, and
  ➢ the transfer time per (typically four-byte) word, which is determined by the physical bandwidth of the communication channel linking the source and destination processors (including data copy effects between buffers)
## Typical parameter values

<table>
<thead>
<tr>
<th>Machine</th>
<th>$t_a$</th>
<th>$t_w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM SP2</td>
<td>40</td>
<td>0.11</td>
</tr>
<tr>
<td>Intel DELTA</td>
<td>77</td>
<td>0.54</td>
</tr>
<tr>
<td>Intel Paragon</td>
<td>121</td>
<td>0.07</td>
</tr>
<tr>
<td>Meiko CS-2</td>
<td>87</td>
<td>0.08</td>
</tr>
<tr>
<td>nCUBE-2</td>
<td>154</td>
<td>2.4</td>
</tr>
<tr>
<td>Thinking Machines CM-5</td>
<td>82</td>
<td>0.44</td>
</tr>
<tr>
<td>Workstations on Ethernet</td>
<td>1500</td>
<td>5.0</td>
</tr>
<tr>
<td>Workstations on FDDI</td>
<td>1150</td>
<td>1.1</td>
</tr>
</tbody>
</table>

![Graph showing message size vs. time for different networks](image)

The graph shows the time (in microseconds) for messages of varying sizes (in bytes) on different networks: Ethernet, FDDI, Paragon, and SP1.
Idle time: basics

- A processor may be idle due to lack of computation or lack of data.

- In the first case, idle time may be avoided by using load-balancing techniques.

- In the second case, the processor is idle while the computation and communication required to generate remote data are performed. This idle time can sometimes be avoided by structuring a program so that processors perform other computation or communication while waiting for remote data.

- This technique is referred to as *overlapping computation and communication*, since local computation is performed concurrently with remote communication and computation.
• Such overlapping can be achieved in two ways
  
  > Creating multiple tasks on each processor so that when one task blocks waiting for remote data, execution may be able to switch to another task for which data are already available (related to partitioning and agglomeration)

  > Alternatively, a single task can be structured so that requests for remote data are interleaved explicitly with other computation ………. here comes out the *synchronization issue*
Synchronization approaches

• Data dependency requires different tasks to synchronize with each other in order to achieve correct execution

• There are two main approaches to address the synchronization problem

  ➢ **Conservative:** any computation goes on only when there is safety on no violation of data dependency

    ✓ Requires safety detection mechanisms to be embedded within the parallel implementation

  ➢ **Optimistic:** computation can proceed even with no safety assurance – data dependency violations are recovered dynamically

    ✓ Requires log/recovery mechanisms to be embedded within the parallel implementation
Synchronization effects

- Conservative synchronization tends to reduce computation time at the expense of idle time

- Optimistic synchronization tends to reduce idle time at the expense of computation time

- Both affect and are affected by communication time
  - Reducing communication cost might increase idle time in case of conservative synchronization
    - At worst the application might deadlock
  - Reducing the communication cost might increase computation time in optimistic synchronization
    - At worst the application might trashing
Explicit reduction of communication time: “primary” message aggregation effects

- Use of message aggregation techniques
- A set of application messages are sent via the same physical message
- Aggregation gain: reduction of communication cost per byte sent
- Aggregation loss: computation and/or idle time increase due to delay in the delivery of application data
- Typically there is the need for a mechanism controlling the effects of aggregation at run-time in any case we cannot determine the corresponding dependence in the performance model
Back to the graph reduction example

• A Variable Age Based (VAB) aggregation technique is used
• Each process Pi collects application messages destined to the same remote process Pj into an aggregation buffer OUT_BUFFi-j
• The module controlling the aggregation keeps an age estimate for each aggregation buffer
• Anytime the messages aggregated in OUT_BUFFi-j are sent, the message rate achieved by the aggregate is calculated
• This rate is used to determine what the maximum age for the next aggregate should be
• If the arrival rate for the current aggregate increases, the maximum age for the next aggregate into that buffer is increased since the application is likely to start a period of bursty exchange of application messages from Pi to Pj
Speedup and efficiency

- Execution time is not always the most convenient metric by which to evaluate parallel algorithm performance
- This is because execution time tends to vary with problem size
- Hence, execution times must be normalized when comparing algorithm performance at different problem sizes
- The following two metrics are typically more representative:
  - **Speedup** – ratio between execution time on one processor and parallel execution time. It characterizes the real acceleration via the parallel implementation
  - **Efficiency** – fraction of time that processors spend doing useful work. It characterizes the effectiveness with which an algorithm uses the computational resources of a parallel computer in a way that is independent of problem size
• Speedup and efficiency can be either relative
  ➢ *Relative* – achieved using as the reference the execution time on one processor of that same parallel algorithm
  ➢ *Absolute* – achieved using as the reference the execution time on one processor of the best known sequential algorithm

**Relation between speedup and efficiency**

\[
\text{Speedup} = P \times \text{Efficiency} = P \times \frac{T_{\text{sequential}}}{(P \times T_{\text{parallel}})}
\]
Scalability with fixed problem size

• An important aspect of performance analysis is the study of how algorithm performance varies with parameters such as problem size and processor count.

• We may evaluate the scalability of a parallel algorithm in terms how effectively it can use an increased number of processors.

• One approach to quantifying scalability is to determine how execution time $T$ and efficiency $E$ vary with increasing processor count $P$ for a fixed problem size and machine parameters.

• This fixed problem analysis allows us to answer questions such as, What is the fastest I can solve problem $A$ on computer $X$? and What is the greatest number of processors I can utilize if I want to maintain an efficiency of 50 percent?

• The latter question may be of interest if a computer is shared and there is a charge for each processor used.
• **Note:** it is important to consider both $E$ and $T$ when evaluating scalability

  ➢ While $E$ will generally decrease monotonically with $P$, $T$ may actually increase if the performance model includes a term proportional to a positive power of $P$

  ➢ In such cases, it may not be productive to use more than some maximum number of processors for a particular problem size and choice of machine parameters
Scalability with scaled problem size

• Large parallel computers are frequently used not only to solve fixed-size problems faster, but also to solve larger problems.

• This observation encourages a different approach to the analysis of algorithms called *scaled problem* analysis, whereby we consider not how $E$ varies with $P$, but how the amount of computation performed must scale with $P$ to keep $E$ constant.

• This function of $N$ is called an algorithm's *isoefficiency function* and can provide valuable insights into algorithm behavior.

• An algorithm with an isoefficiency function of $O(P)$ is highly scalable, since the amount of computation needs to increase only linearly with respect to $P$ to keep efficiency constant.

• In contrast, an algorithm with a quadratic or exponential isoefficiency function would be poorly scalable.
• Given that $E = \frac{T_1}{T_{\text{comp}} + T_{\text{comm}} + T_{\text{idle}}}$ to maintain constant efficiency $E$, uniprocessor time must increase at the same rate as total parallel time.

• Equivalently, the amount of essential computation must increase at the same rate as overheads due to replicated computation, communication, and idle time.

• Scaled problem analysis does not make sense for all problems:
  - Real-time constraints, e.g., in weather forecasting, may require that computation be completed in a fixed amount of time.
  - In other applications, scaling is not possible because of physical constraints on problem size. For example, in molecular modeling, the number of atoms in a molecule is fixed.
Execution profiles

• If scalability analysis suggests that performance is poor on problem sizes and computers of interest, we can use models to identify likely sources of inefficiency and hence areas in which an algorithm can be improved

• Poor performance may be due to excessive replicated computation, idle time, message startups, data transfer costs, or some combination of these factors

• An important first step when attempting to improve an algorithm is to identify which of these factors is dominant

• One can do this by computing an expected execution profile for the algorithm, indicating the contributions of these different factors to execution time as a function of $N$ and/or $P$
One example
Evaluation vs benchmarking

• Sometimes it is not possible or not useful to express the speedup analytically

• Instead, an experimental evaluation of the parallel algorithm is performed to establish its effectiveness

• This might be the case of parallel platforms designed for classes of problems, whose run time behavior on a given system is affected by the specific problem instance (not only by the problem size)

• In such a case, some problem instances are used to evaluate the parallel platform

• This is the case of the previously presented graph reductor
Results for benchmark graph A (Church representation of the forth power of four)
Results for benchmark graph B (iterated exponential ELL term)
intrinsically sequential phase
Amdahl's law

• Common observation regarding parallel processing is that every algorithm has a sequential component that will eventually limit the speedup that can be achieved on a parallel computer

• This observation is often codified as Amdahl's law, which can be stated as follows: if the sequential component of an algorithm accounts for $1/s$ of the program's execution time, then the maximum possible speedup that can be achieved on a parallel computer is $s$

• For example, if the sequential component is 5 percent, then the maximum speedup that can be achieved is 20
Parallel programming environments: MPI
The MPI programming model

• In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes.

• In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as multiple program multiple data (MPMD) to distinguish it from the SPMD model in which every processor executes the same program.

• Because the number of processes in an MPI computation is normally fixed, our focus is on the mechanisms used to communicate data between processes.
• Processes can use \textit{point-to-point} communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications.

• A group of processes can call \textit{collective} communication operations to perform commonly used global operations such as summation and broadcast.

• MPI's ability to \textit{probe} for messages supports asynchronous communication.

• Probably MPI's most important feature from a software engineering viewpoint is its support for modular programming. A mechanism called a \textit{communicator} allows the MPI programmer to define modules that encapsulate internal communication structures.
MPI basics

• MPI basic functions

MPI_INIT : Initiate an MPI computation.
MPI_FINALIZE : Terminate a computation.
MPI_COMM_SIZE : Determine number of processes.
MPI_COMM_RANK : Determine my process identifier.
MPI_SEND : Send a message.
MPI_RECV : Receive a message.
MPI_INIT(int *argc, char ***argv)
  Initiate a computation.
  
  argc, argv are required only in the C language binding,
  where they are the main program's arguments.

MPI_FINALIZE()
  Shut down a computation.

MPI_COMM_SIZE(comm, size)
  Determine the number of processes in a computation.
  
  IN   comm    communicator (handle)
  OUT  size    number of processes in the group of comm (integer)

MPI_COMM_RANK(comm, pid)
  Determine the identifier of the current process.
  
  IN   comm    communicator (handle)
  OUT  pid     process id in the group of comm (integer)
MPI_SEND(buf, count, datatype, dest, tag, comm)
Send a message.

IN buf address of send buffer (choice)
IN count number of elements to send (integer ≥ 0)
IN datatype datatype of send buffer elements (handle)
IN dest process id of destination process (integer)
IN tag message tag (integer)
IN comm communicator (handle)

MPI_RECV(buf, count, datatype, source, tag, comm, status)
Receive a message.

OUT buf address of receive buffer (choice)
IN count size of receive buffer, in elements (integer ≥ 0)
IN datatype datatype of receive buffer elements (handle)
IN source process id of source process, or MPI_ANY_SOURCE (integer)
IN tag message tag, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (status)
MPI communicators

• A communicator identifies the process group and context with respect to which the operation is to be performed.

• Communicators provide a mechanism for identifying process subsets during development of modular programs and for ensuring that messages intended for different purposes are not confused.

• In most cases, it suffices to provide the default value MPI_COMM_WORLD, which identifies all processes involved in a computation.
program main
begin

MPI_INIT() Initiate computation

MPI_COMM_SIZE(MPI_COMM_WORLD, count) Find # of processes

MPI_COMM_RANK(MPI_COMM_WORLD, myid) Find my id

print("I am", myid, "of", count) Print message

MPI_FINALIZE() Shut down

end
C language binding

• In the C language binding, function names are as in the MPI definition but with only the MPI prefix and the first letter of the function name in upper case.

• Status values are returned as integer return codes.

• The return code for successful completion is MPI_SUCCESS; a set of error codes is also defined.

• Compile-time constants are all in upper case and are defined in the file mpi.h, which must be included in any program that makes MPI calls.

• Handles are represented by special defined types, defined in mpi.h.
• Function parameters with type IN are passed by value, while parameters with type OUT and INOUT are passed by reference (that is, as pointers).

• A status variable has type MPI_Status and is a structure with fields status.MPI_SOURCE and status.MPI_TAG containing source and tag information.

• An MPI datatype is defined for each C datatype:
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_UNSIGNED_CHAR
  - MPI_UNSIGNED
  - MPI_UNSIGNED_LONG
  - MPI_FLOAT, MPI_DOUBLE
  - MPI_LONG_DOUBLE, etc.
Determinism vs non-determinism

- Message-passing programming models are by default nondeterministic: the arrival order of messages sent from two processes, A and B, to a third process, C, is not defined.

- However, MPI does guarantee that two messages sent from one process, A, to another process, B, will arrive in the order sent.

- It is the programmer's responsibility to ensure that a computation is deterministic when (as is usually the case) this is required.

- In the task/channel programming model, determinism is guaranteed by defining separate channels for different communications and by ensuring that each channel has a single writer and a single reader. Hence, a process C can distinguish messages received from A or B as they arrive on separate channels.
• MPI does not support channels directly, but it allows a receive operation to specify a source, tag, and/or context. (Recall that these data constitute a message's envelope.)

• The source specifier in the MPI_RECV function allows the programmer to specify that a message is to be received either from a single named process (specified by its integer process identifier) or from any process (specified by the special value MPI_ANY_SOURCE).

• Message tags provide a further mechanism for distinguishing between different messages. A sending process must associate an integer tag with a message. This is achieved via the tag field in the MPI_SEND call. A receiving process can then specify that it wishes to receive messages either with a specified tag or with any tag (MPI_ANY_TAG).
Global operations

• For convenience, and to permit optimized implementations, MPI also provides a suite of specialized collective communication functions that perform commonly used operations of this type.

• These functions include the following.
  - Barrier: Synchronizes all processes
  - Broadcast: Sends data from one process to all processes
  - Gather: Gathers data from all processes to one process
  - Scatter: Scatters data from one process to all processes
  - Reduction operations: Sums, multiplies, etc., distributed data
**MPI_Barrier(comm)**

Global synchronization.

*IN* comm communicator (handle)

**MPI_Bcast(inbuf, incnt, intype, root, comm)**

Broadcast data from root to all processes.

*INOUT* inbuf address of input buffer, or output buffer at root (choice)

*IN* incnt number of elements in input buffer (integer)

*IN* intype datatype of input buffer elements (handle)

*IN* root process id of root process (integer)

*IN* comm communicator (handle)

**MPI_Gather(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**

**MPI_Scatter(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)**

Collective data movement functions.

*IN* inbuf address of input buffer (choice)

*IN* incnt number of elements sent to each (integer)

*IN* intype datatype of input buffer elements (handle)

*OUT* outbuf address of output buffer (choice)

*IN* outcnt number of elements received from each (integer)

*IN* outtype datatype of output buffer elements (handle)

*IN* root process id of root process (integer)

*IN* comm communicator (handle)
All these operations are executed collectively, meaning that each process in a process group calls the communication routine with the same parameters.
Details

- MPI_BARRIER is used to synchronize execution of a group of processes. No process returns from this function until all processes have called it.

- MPI_BCAST, MPI_GATHER, and MPI_SCATTER are collective data movement routines, in which all processes interact with a distinguished root process to broadcast, gather, or scatter data, respectively. In each case, the first three arguments specify the location (inbuf) and type (intype) of the data to be communicated and the number of elements to be sent to each destination (incnt). Other arguments specify the location and type of the result (outbuf, outtype) and the number of elements to be received from each source (outcnt).
• MPI_BCAST implements a one-to-all broadcast operation whereby a single named process (root) sends the same data to all other processes; each process receives this data from the root process. At the time of call, the data are located in inbuf in process root and consists of incnt data items of a specified intype. After the call, the data are replicated in inbuf in all processes. As inbuf is used for input at the root and for output in other processes, it has type INOUT.

• MPI_GATHER implements an all-to-one gather operation. All processes (including the root process) send data located in inbuf to root. This process places the data in contiguous nonoverlapping locations in outbuf, with the data from process $i$ preceding that from process $i+1$. Hence, the outbuf in the root process must be $P$ times larger than inbuf, where $P$ is the number of processes participating in the operation. The outbuf in processes other than the root is ignored.
MPI_SCATTER implements a one-to-all *scatter* operation; it is the reverse of MPI_GATHER. A specified root process sends data to all processes, sending the *i* th portion of its inbuf to process *i*; each process receives data from root in outbuf. Hence, the inbuf in the root process must be *P* times larger than outbuf. Notice the subtle difference between this function and MPI_BCAST: while in MPI_BCAST every process receives the *same* value from the root process, in MPI_SCATTER every process receives a *different* value.
Processes 

Data

One-to-all broadcast

MPI_BCAST

All-to-one gather

MPI_GATHER

One-to-all scatter

MPI_SCATTER
• The functions MPI_REDUCE and MPI_ALLREDUCE implement reduction operations. They combine the values provided in the input buffer of each process, using a specified operation op, and return the combined value either to the output buffer of the single root process (in the case of MPI_REDUCE) or to the output buffer of all processes (MPI_ALLREDUCE). The operation is applied pointwise to each of the count values provided by each process. All operations return count values with the same datatype as the operands. Valid operations include maximum and minimum (MPI_MAX and MPI_MIN); sum and product (MPI_SUM and MPI_PROD); logical and, or, and exclusive or (MPI_LAND, MPI_LOR, and MPI_LXOR); and bitwise and, or, and exclusive or (MPI_BAND, MPI_BOR, and MPI_BXOR)
Processes . . .

Initial Data:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4</td>
<td>5 7</td>
<td>0 3</td>
<td>6 2</td>
</tr>
</tbody>
</table>

MPI_REDUCE with MPI_MIN, root = 0:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

MPI_ALLREDUCE with MPI_MIN:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2</td>
<td>0 2</td>
<td>0 2</td>
<td>0 2</td>
</tr>
</tbody>
</table>

MPI_REDUCE with MPI_SUM, root = 1:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>13 16</td>
<td>--</td>
</tr>
</tbody>
</table>
Asynchronous communication

- Asynchronous communication is supported by the MPI_IPROBE, MPI_PROBE, and MPI_GET_COUNT functions.

```c
MPI_IPROBE(source, tag, comm, flag, status)
Poll for a pending message.
IN    source  id of source process, or MPI_ANY_SOURCE (integer)
IN    tag      message tag, or MPI_ANY_TAG (integer)
IN    comm     communicator (handle)
OUT   flag     (logical/Boolean)
OUT   status   status object (status)
```

```c
MPI_PROBE(source, tag, comm, status)
Return when message is pending.
IN    source  id of source process, or MPI_ANY_SOURCE (integer)
IN    tag      message tag, or MPI_ANY_TAG (integer)
IN    comm     communicator (handle)
OUT   status   status object (status)
```

```c
MPI_GET_COUNT(status, datatype, count)
Determine size of a message.
IN    status   status variable from receive (status)
IN    datatype  datatype of receive buffer elements (handle)
OUT   count    number of data elements in message (integer)
```
An example

- The following code fragment uses these functions to receive a message from an unknown source and containing an unknown number of integers.

- It first detects the arrival of the message using MPI_PROBE. Then, it determines the message source and uses MPI_GET_COUNT to determine the message size. Finally, it allocates a buffer of the appropriate size and receives the message.

```c
int count, *buf, source;
MPI_Probe(MPI_ANY_SOURCE, 0, comm, &status);
source = status.MPI_SOURCE;
MPI_Get_count(status, MPI_INT, &count);
buf = malloc(count*sizeof(int));
MPI_Recv(buf, count, MPI_INT, source, 0, comm, &status);
```