Parallel Algorithm Analysis and Design

CPS343

Parallel and High Performance Computing

Spring 2018

Outline

Acknowledgements

Material used in creating these slides comes from "Designing and Building Parallel Programs" by Ian Foster, Addison-Wesley, 1995. Available on-line at `http://www.mcs.anl.gov/~itf/dbpp/`

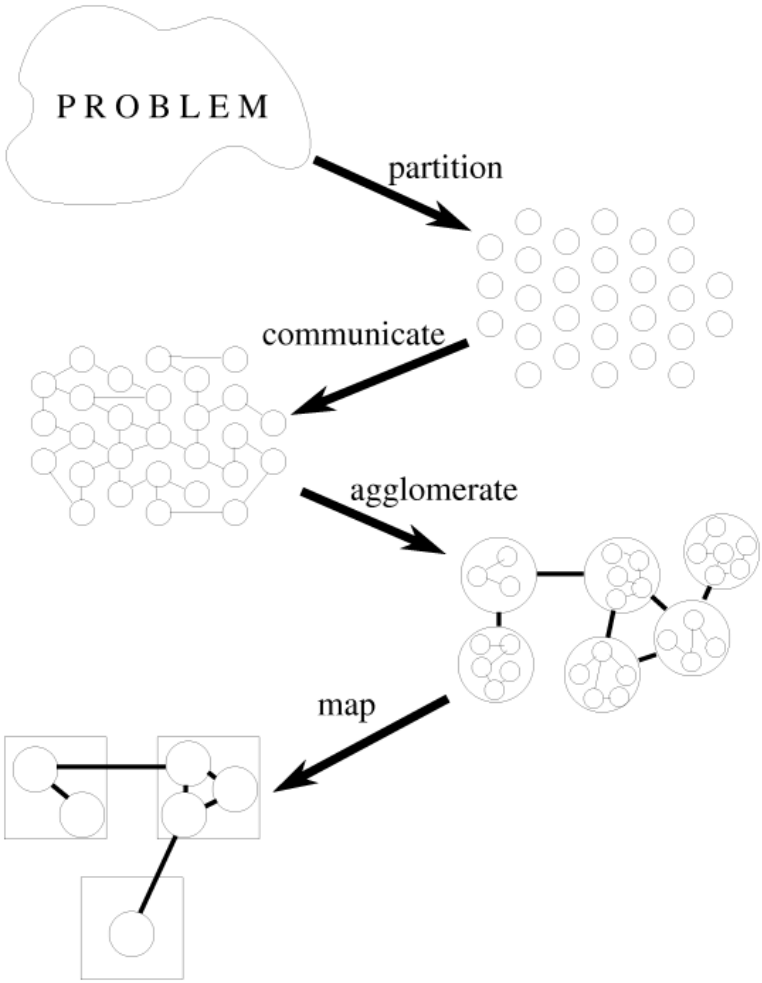
Outline

Foster's model

In *"Designing and Building Parallel Programs"* Foster proposes a model with tasks that interact with each other by communicating through channels.

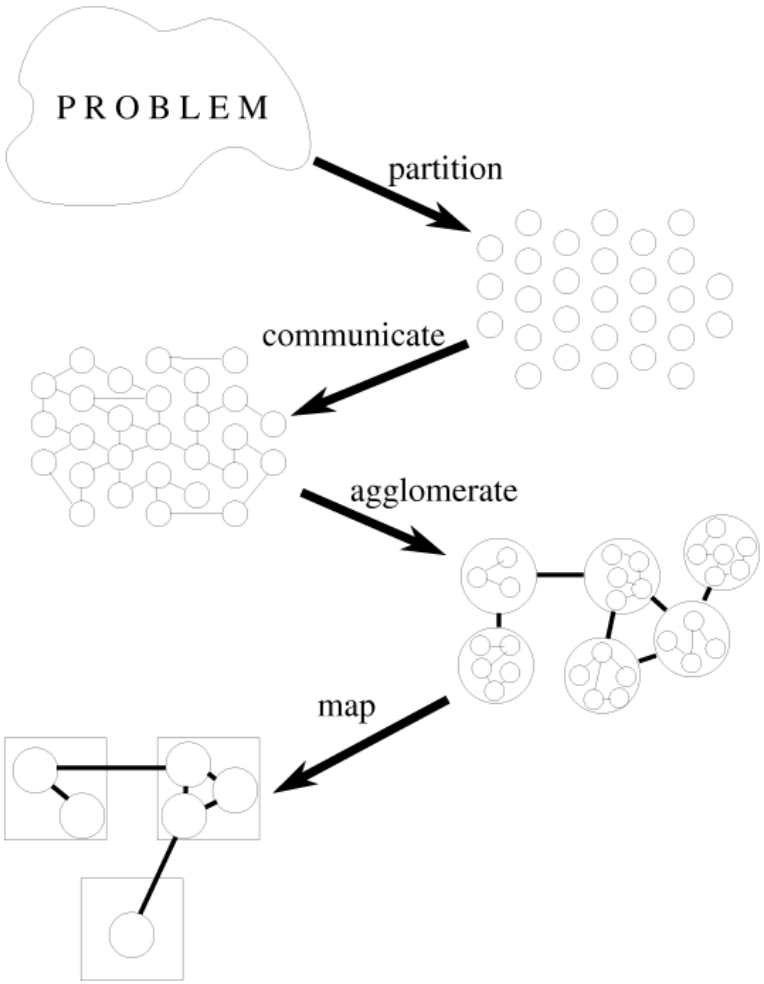
- A **task** is a program, its local memory, and its communication *inports* and *outports*.
- A **channel** connects a task's inport to another task's outport.
- Channels are buffered. Sending is **asynchronous** while receiving is **synchronous** (receiving task is blocked until expected message arrives).

Four-phase design process: PCAM



- **Partitioning**. The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
- **Communication**. The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

Four-phase design process: PCAM



- **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.
- **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.

Outline

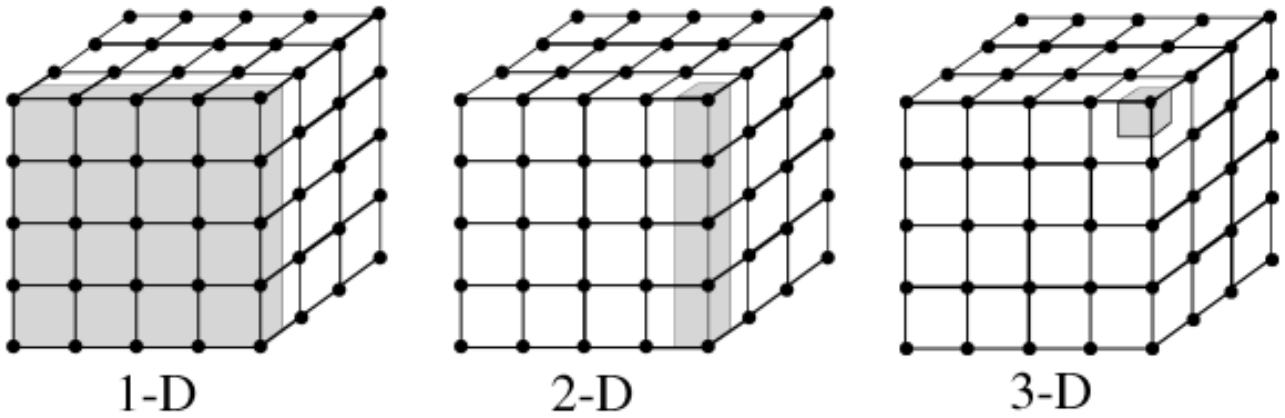
Partitioning

- The partitioning stage of a design is intended to *expose opportunities* for parallel execution.
- Focus is on defining a *large number of small tasks* (fine-grained decomposition).
- A good partition divides *both the computation and the data* into small pieces.
- One approach is to focus first on partitioning the data associated with a problem; this is called *domain decomposition*.
- The alternative approach, termed *functional decomposition*, decomposes the computation into separate tasks before considering how to partition the data.
- These are complementary techniques.
- Seek to avoid replicating computation and data (may change this later in process).

Domain decomposition

- First partition data; ideally divide data into small pieces of approximately equal size.
- Next partition computation, typically by associating each operation with the data on which it operates.
- Focus first on the largest data structure or on the data structure that is accessed most frequently.

Domain decomposition example: 3-D cube of data



- 1-D decomposition: split cube into a 1-D array of slices (each slice is 2-D, coarse granularity)
- 2-D decomposition: split cube into a 2-D array of columns (each column is 1-D)
- 3-D decomposition: split cube into a 3-D array of individual data elements. (fine granularity)

Functional decomposition

- Initial focus is on the computation that is to be performed rather than on the data.
- Divide computation into disjoint tasks.
- Examine data requirements of tasks:
  1. Requirements may be disjoint, in which case the partition is complete.
  2. Requirements may overlap significantly, in which case considerable communication will be required to avoid replication of data.
  3. Second case is a sign that a domain decomposition approach should be considered instead.

Functional decomposition

- Functional decomposition is valuable as a different way of thinking about problems and should be considered when exploring possible parallel algorithms.
- A focus on the computations that are to be performed can sometimes reveal structure in a problem, and hence opportunities for optimization, that would not be obvious from a study of data alone.
- Functional decomposition is an important program structuring technique; can reduce the complexity of the overall design.

Partitioning design checklist

Questions to consider before finishing the partitioning step:

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer?
2. Does your partition avoid redundant computation and storage requirements?
3. Are tasks of comparable size?
4. Does the number of tasks scale with problem size?
5. Have you identified several alternative partitions?

Outline

Communication

- Conceptualize a need for communication between two tasks as a *channel linking the tasks*, on which one task can send messages and from which the other can receive.
- Channel structure links tasks that require data (consumers) with tasks that possess those data (producers).
- Definition of a channel involves an intellectual cost and the sending of a message involves a physical cost — *avoid introducing unnecessary channels and communication operations.*
- We want to distribute communication operations over many tasks.
- We want to organize communication operations in a way that permits concurrent execution.

Communication in domain and functional decomposition

Communication requirements can be difficult to determine in domain decomposition problems.

- First partition data structures into disjoint subsets and then associate with each datum those operations that operate solely on that datum.
- Often there are operations that require data from several tasks; these must be dealt with separately.
- Organizing the resulting communication in an efficient manner can be challenging.

Communication requirements in parallel algorithms obtained by functional decomposition are often straightforward as they usually correspond to the data flow between tasks.

Patterns of communication

Foster categorizes communication patterns along four loosely orthogonal axes:

1. **local** $\leftrightarrow$ **global**
2. structured $\leftrightarrow$ unstructured
3. static $\leftrightarrow$ dynamic
4. synchronous $\leftrightarrow$ asynchronous

local: each task communicates with a small set of other tasks.

global: requires each task to communicate with many tasks.

Patterns of communication

Foster categorizes communication patterns along four loosely orthogonal axes:

1. local $\leftrightarrow$ global
2. **structured** $\leftrightarrow$ **unstructured**
3. static $\leftrightarrow$ dynamic
4. synchronous $\leftrightarrow$ asynchronous

structured: a task and its neighbors form a regular structure, such as a tree or grid.

unstructured: networks may be arbitrary graphs.

Patterns of communication

Foster categorizes communication patterns along four loosely orthogonal axes:

1. local $\leftrightarrow$ global
2. structured $\leftrightarrow$ unstructured
3. **static** $\leftrightarrow$ **dynamic**
4. synchronous $\leftrightarrow$ asynchronous

static: the identity of communication partners does not change over time.

dynamic: the identity of communication partners may be determined by data computed at runtime and may be highly variable.

Patterns of communication

Foster categorizes communication patterns along four loosely orthogonal axes:

1. local $\leftrightarrow$ global
2. structured $\leftrightarrow$ unstructured
3. static $\leftrightarrow$ dynamic
4. **synchronous** $\leftrightarrow$ **asynchronous**

synchronous: producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations.

asynchronous: may require a consumer to receive data without the cooperation of the producer.

Local communication

- A *local communication structure* is obtained when an operation requires data from a small number of other tasks.
- Easy to define channels that link consumer task (needs the data) with the producer tasks (have the data).
- Example: Finite differences with Jacobi iteration.

Local communication: Jacobi finite differences

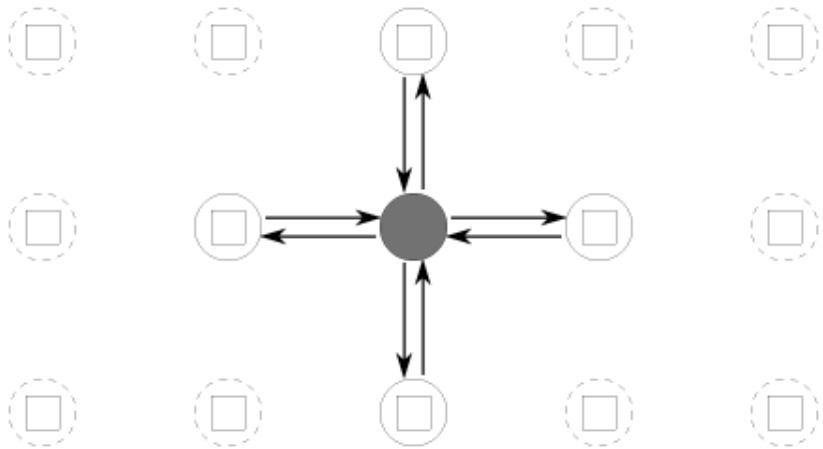
- Finite differences is a method used to solve certain differential equation problems.
- In the Jacobi iteration, a multidimensional grid is repeatedly updated by replacing the value at each point with a weighted average of the values at a small, fixed number of neighboring points.
- Set of values required to update a single grid point is called that grid point's *stencil.*
- For example,

$$X_{i,j}^{(t+1)} = \frac{X_{i-1,j}^{(t)} + X_{i,j-1}^{(t)} + 4X_{i,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j+1}^{(t)}}{8}$$

uses a five-point stencil to update each element $X_{i,j}$ of a two-dimensional grid X. The variable t indicates the time step and i and j denote the grid locations.

Local communication: Jacobi finite differences

The communications channels for a particular node are shown by the arrows in the diagram on the right.



- Assume that the domain decomposition results in a distinct task for each point in the two-dimensional grid. The task allocated to $X_{i,j}$ must compute the sequence $X_{i,j}^{(1)}, X_{i,j}^{(2)}, X_{i,j}^{(3)}, \ldots$
- This computation requires in turn the four corresponding sequences which are produced by the four neighboring tasks:

$$X_{i-1,j}^{(1)}, X_{i-1,j}^{(2)}, X_{i-1,j}^{(3)}, \ldots, \quad X_{i,j-1}^{(1)}, X_{i,j-1}^{(2)}, X_{i,j-1}^{(3)}, \ldots,$$
$$X_{i+1,j}^{(1)}, X_{i+1,j}^{(2)}, X_{i+1,j}^{(3)}, \ldots, \quad X_{i,j+1}^{(1)}, X_{i,j+1}^{(2)}, X_{i,j+1}^{(3)}, \ldots$$

Local communication: Jacobi finite differences

- Define channels linking each task that requires a value with the task that generates that value.
- Each task then executes the following logic:

for $t = 0$ to $T - 1$
    send $X_{i,j}^{(t)}$ to each neighbor
    receive $X_{i-1,j}^{(t)}$, $X_{i,j-1}^{(t)}$, $X_{i+1,j}^{(t)}$, and $X_{i,j+1}^{(t)}$ from neighbors
    compute $X_{i,j}^{(t+1)}$
endfor

Global communication

- In contrast to local communication, a *global communication operation* is one in which many tasks must participate.
- When such operations are implemented, it may not be sufficient simply to identify individual producer/consumer pairs.
- May result in too many communications or may restrict opportunities for concurrent execution.

Global communication

- Consider a *parallel reduction* operation, that is, an operation that reduces N values distributed over N tasks using a commutative associative operator such as addition: $S = \sum X_i$.
- If a single "manager" task requires the result S we can define a communication structure that allows each task to communicate its value to the manager independently.
- Because the manager can receive and add only one number at a time, this approach takes $O(N)$ time to sum N numbers—not a very good parallel algorithm!

Global communication

- Two general problems that can prevent efficient parallel execution of an algorithm:
  1. The algorithm is *centralized:* it does not distribute computation and communication. A single task (in this case, the manager task) must participate in every operation.
  2. The algorithm is *sequential:* it does not allow multiple computation and communication operations to proceed concurrently.
- Both of these problems must be addressed to develop a good parallel algorithm.

Distributing communication and computation

One way we can distribute the summation of the N numbers is by making each task i, where $0 < i < N - 1$, compute the sum: $S_i = X_i + S_{i-1}$:

- Communication requirements associated with this algorithm can be satisfied by connecting the N tasks in a one-dimensional array.
- Task $N - 1$ sends its value to its neighbor in this array.
- Tasks 1 through $N - 2$ each wait to receive a partial sum from their right-hand neighbor, add this to their local value, and send the result to their left-hand neighbor.
- Task 0 receives a partial sum and adds this to its local value to obtain the complete sum.

Distributing communication and computation

- This approach distributes the $N-1$ communications and additions, but is still sequential if only a single sum is required.
- If multiple multiple summation operations are to be performed then parallelism can configuring the array of tasks as a pipeline, through which flow partial sums.
- Each summation still takes $N-1$ steps, but if there are multiple sums, many of these steps can be overlapped.

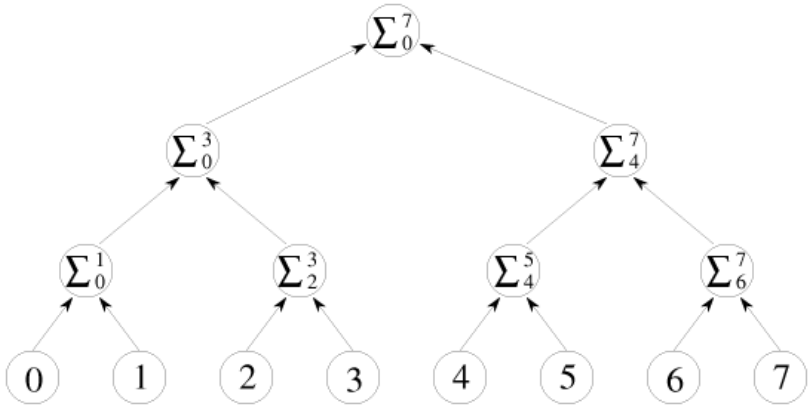
Uncovering concurrency: Divide and conquer

- To parallelize this problem for a single summation, we can to partition it into two or more simpler problems of roughly equivalent size (e.g., summing $N/2$ numbers).
- This process is applied recursively to produce a set of subproblems that cannot be subdivided further (e.g., summing two numbers).
- This divide-and-conquer technique is effective in parallel computing when the subproblems generated by problem partitioning can be solved concurrently.

Divide and conquer algorithm

```
divide_and_conquer()
    if base case then
        solve problem
    else
        partition problem into subproblems L and R
        solve problem L using divide_and_conquer()
        solve problem R using divide_and_conquer()
        combine solutions to problems L and R
    endif
end
```

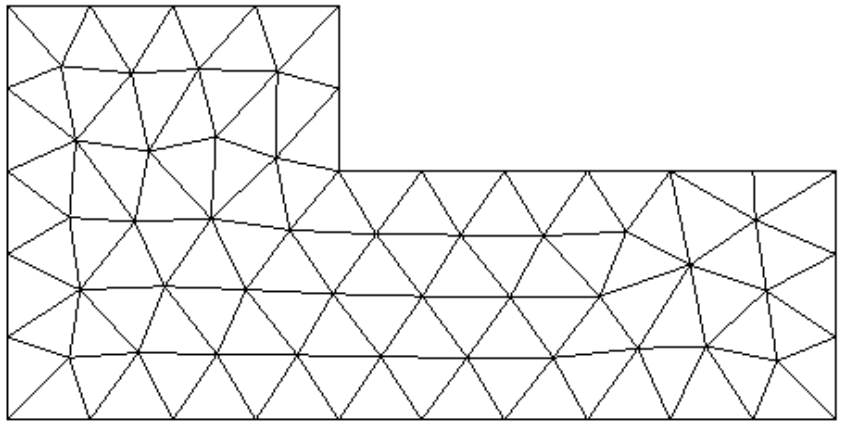
Divide and conquer analysis



- Assuming that N is a power of 2, the decomposition can be carried out until the base problem is the sum of two numbers.
- The operations on each level can be done simultaneously, so the summation can be carried out in $\log N$ steps rather than N steps.
- If N is not a power of 2 then the operation requires $\lceil \log N \rceil$ steps.
- We have distributed the $N - 1$ communication and computation operations required to perform the summation, and
- We have modified the order in which these operations are performed so that they can proceed concurrently.

Unstructured and dynamic communication

- Foster's example comes from finite elements, where the finite element mesh is composed of triangles and the number of edges incident to a vertex is not constant.



- Channel structure representing communication partners can irregular, data-dependent and can change over time.
- Unstructured communication complicates the tasks of agglomeration and mapping.
- It is often nontrivial to determine an agglomeration strategy that both creates tasks of nearly equal size and minimizes communication requirements by creating the least number of intertask edges.

Image source: http://atlas.gcsc.uni-frankfurt.de/~ug/ddd/tutorial/fe.html

Asynchronous communication

- In this case, tasks that possess data (producers) are not able to determine when other tasks (consumers) may require data
- Consumers must explicitly request data from producers

Communication checklist

Questions to consider before finishing the communication analysis step:

1. Do all tasks perform about the same number of communication operations?
2. Does each task communicate only with a small number of neighbors?
3. Are communication operations able to proceed concurrently?
4. Is the computation associated with different tasks able to proceed concurrently?

Outline

Agglomeration

- At this point we've broken down our problem enough that we understand the individual tasks and the necessary communication between tasks.
- The goal now is to being making the parallel solution practical and as efficient as possible.
- There are two main questions:
  1. is it useful to combine, or agglomerate, tasks to reduce the number of tasks?
  2. is it worthwhile to replicate data and/or computation?
- The number of tasks yielded by the agglomeration phase, although reduced, may still be greater than the number of processors. Resolution is deferred to the mapping phase.

Agglomeration: Conflicting goals

Three sometimes-conflicting goals guide decisions concerning agglomeration and replication:

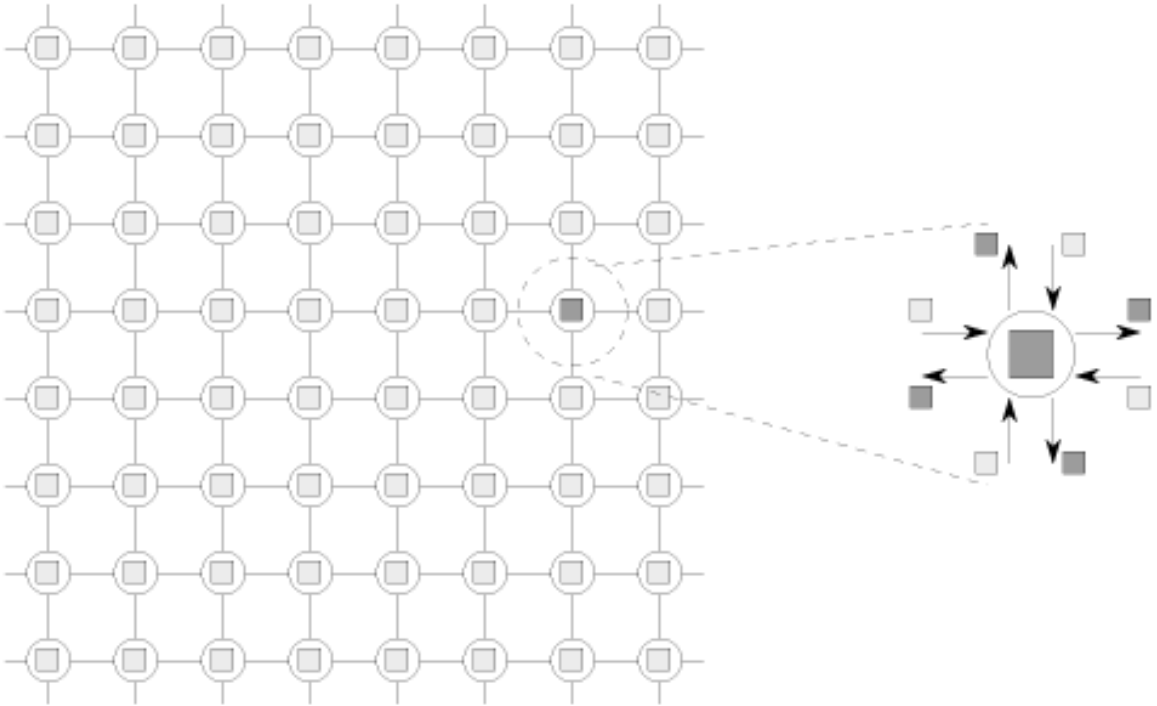
1. reducing communication costs by increasing computation and communication granularity,
2. retaining flexibility with respect to scalability and mapping decisions, and
3. reducing software engineering costs.

Increasing granularity

- A large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.
- Communication costs and task creation costs are overhead that can be reduced by increasing granularity.

Increasing granularity: Fine grained version

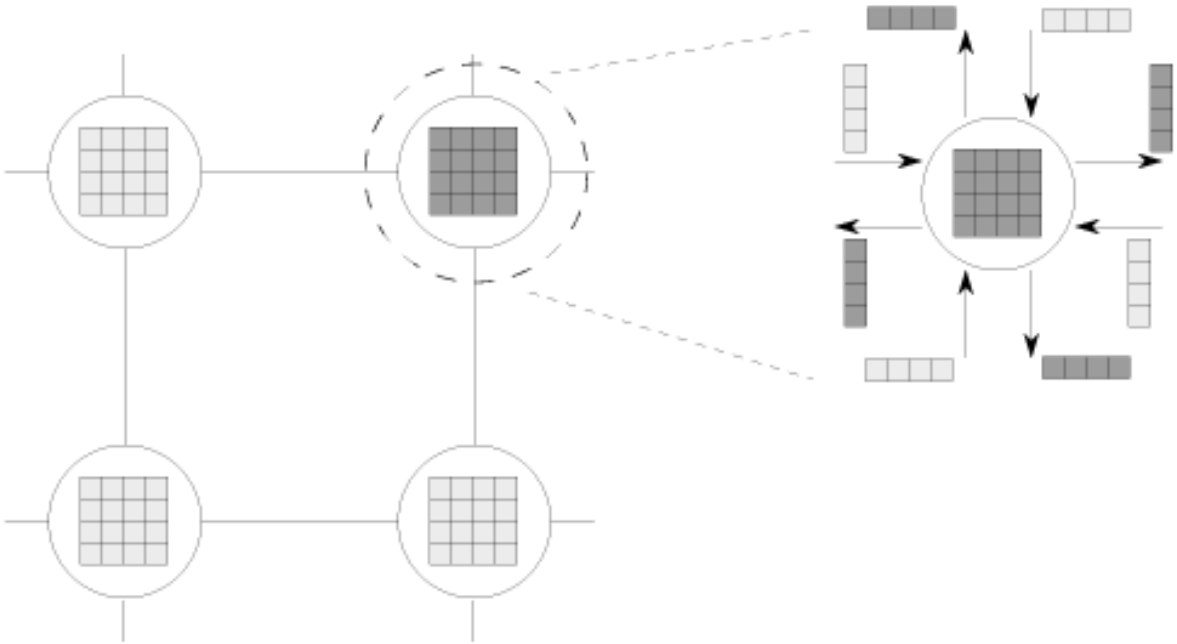
- Fine-grained partition of 8×8 grid.
- Partitioned into 64 tasks.
- Each task responsible for a single point.
- $64 \times 4 = 256$ communications are required, 4 per task.
- Total of 256 data values transferred.



Outgoing messages are dark shaded and incoming messages are light shaded.

Increasing granularity: Coarse grained version

- Coarse-grained partition of 8×8 grid.
- Partitioned into 4 tasks.
- Each task responsible for 16 points.
- $4 \times 4 = 16$ communications are required.
- total of $16 \times 4 = 64$ data values transferred.



Outgoing messages are dark shaded and incoming messages are light shaded.

Surface-to-volume effects

- This reduction in communication costs is due to a *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.
- In a two-dimensional problem, the "surface" scales with the problem size while the "volume" scales as the problem size squared.
- The *communication/computation ratio* decreases as task size increases.
- 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.
- Designing an efficient agglomeration strategy can be difficult in problems with unstructured communications.

Replicating computation

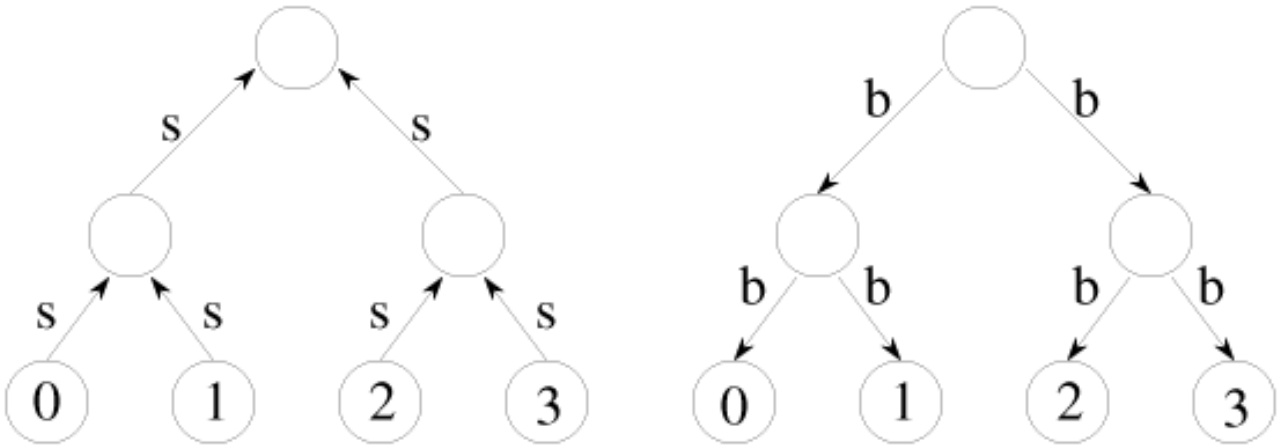
- Sometimes it's more efficient for a task to compute a needed quantity rather than to receive it from another task where it is already known or has been computed.
- Alternatively, sometimes communication and computation can be overlapped to reduce the number of communication cycles necessary to distribute computed data.

Replicating computation example

Sum followed by broadcast: N tasks each have a value that must be combined into a sum and made available to all tasks.

1. Task receives a partial sum from neighbor, updates sum, and passes on updated value. Task 0 completes the sum and sends it back. This requires $2(N-1)$ communication steps.

2. Alternative: Reduction and broadcast sequence that requires only $2 \log N$ communication steps.

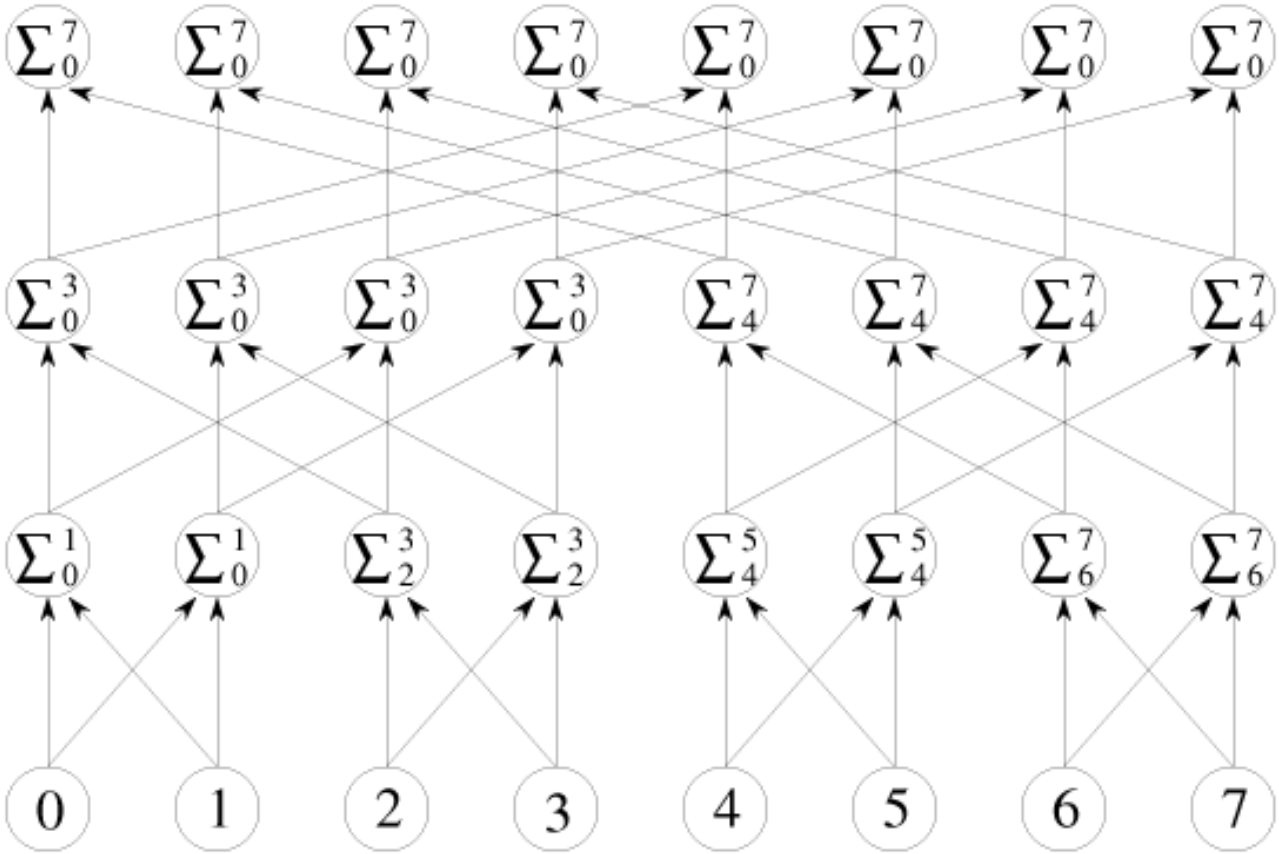


Replicating computation example

- These algorithms are optimal in the sense that they do not perform any unnecessary computation or communication.
- To improve the first summation, assume that tasks are connected in a ring rather than an array, and all N tasks execute the same algorithm so that N partial sums are in motion simultaneously. After $N-1$ steps, the complete sum is replicated in every task.
- This strategy avoids the need for a subsequent broadcast operation, but at the expense of $(N-1)^2$ redundant additions and $(N-1)^2$ unnecessary (but simultaneous) communications.

Replicating computation example

- The tree summation algorithm can be modified so that after $\log N$ steps each task has a copy of the sum. When the communication structure is a *butterfly* structure there are only $O(N \log N)$ operations. In the case that $N = 8$ this looks like:



Avoiding communication

Agglomeration is almost always beneficial if analysis of communication requirements reveals that a set of tasks cannot execute concurrently.

Preserving flexibility

- It is important when agglomerating to avoid making design decisions that limit unnecessarily an algorithm's scalability.
- Don't assume during the design that the number of processors will always be limited to the currently available number.
- Good parallel algorithms are designed to be resilient to changes in processor count.
- It can be advantageous to map several tasks to a processor. Then, a blocked task need not result in a processor becoming idle, since another task may be able to execute in its place.

Reducing software engineering costs

- An additional concern, which can be particularly important when parallelizing existing sequential codes, is the relative development costs associated with different partitioning strategies.
- The most useful strategies may be those that avoid extensive code changes, can make use of existing code, or are required by other constraints (e.g. interface to other software products)

Agglomeration design checklist

Questions to consider before finishing the agglomeration step:

1. Has agglomeration reduced communication costs by increasing locality?
2. If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs, for a range of problem sizes and processor counts?
3. If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?
4. Has agglomeration yielded tasks with similar computation and communication costs?

Agglomeration design checklist (continued)

5. Does the number of tasks still scale with problem size?
6. If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers?
7. Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability?
8. If you are parallelizing an existing sequential program, have you considered the cost of the modifications required to the sequential code?

Outline

Mapping

- At this point we have a set of tasks and we need to assign them to processors on the available machine.
- The mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling.
- General-purpose mapping mechanisms have yet to be developed for scalable parallel computers.
- Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:
  1. We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
  2. We place tasks that communicate frequently on the *same* processor, so as to increase locality.
- The general-case mapping problem is *NP*-complete.

Mapping

- Considerable knowledge has been gained on specialized strategies and heuristics and the classes of problem for which they are effective.
- When domain decomposition is used there is often a fixed number of equal-sized tasks and structured local and global communication.
- If, instead, there are variable amounts of work per task and/or unstructured communication patterns, we might use *load balancing* algorithms that seek to identify efficient agglomeration and mapping strategies.
- The time required to execute these algorithms must be weighed against the benefits of reduced execution time. *Probabilistic load-balancing* methods tend to have lower overhead than do methods that exploit structure in an application.

Mapping

- When either the number of tasks or the amount of computation or communication per task changes dynamically during program execution we might use *dynamic load-balancing* strategy in which a load-balancing algorithm is executed periodically to determine a new agglomeration and mapping.
- If functional decomposition is used we can use *task-scheduling* algorithms which allocate tasks to processors that are idle or that are likely to become idle.
- We will now examine these load-balancing strategies and task scheduling algorithms more carefully.

Load balancing

- Recursive Bisection
- Local Algorithms
- Probabilistic Methods
- Cyclic Methods

Recursive bisection

- Partition a domain into subdomains of approximately equal computational cost while attempting to minimize the number of channels crossing task boundaries.
- 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 bisection

- *Recursive coordinate bisection* — normally applied to irregular grids that have a mostly local communication structure.
  - Cuts made so that grid points in a subdomain all sit on one side of some coordinate boundary.
  - Simple, but does not optimize communication well.
- *Unbalanced recursive bisection* — attempts to reduce communication costs by forming subgrids that have better aspect ratios.
  - Considers the $P-1$ partitions obtained by forming pairs of unbalanced subgrids with $1/P$ and $(P-1)/P$ of the load, with $2/P$ and $(P-2)/P$ of the load, and so on
  - Chooses the partition that minimizes partition aspect ratio.
- *Recursive graph bisection* — uses connectivity information to reduce the number of grid edges crossing subdomain boundaries, and hence to reduce communication requirements.

Local algorithms

- Above techniques are relatively expensive because they require global knowledge of computation state.
- Local load-balancing algorithms compensate for changes in computational load using only information obtained from a small number of neighboring processors.
- Useful in situations in which load is constantly changing but less good at balancing load than global algorithms
- Can be slow to adjust to major changes in load characteristics.

Probabilistic methods

- Allocate tasks to randomly selected processors.
- If the number of tasks is large, we expect that each processor will be allocated about the same amount of computation.
- Advantages are 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 mappings

- Similar to probabilistic methods, but the pattern of task-to-processor assignment is done cyclically and follows some specific enumeration.
- Benefit of improved load balance must be weighed against increased communication costs due to reduced locality.

Task-scheduling algorithms

- Task-scheduling algorithms can be used when a functional decomposition yields many tasks, each with weak locality requirements.
- A centralized or distributed task pool is maintained, into which new tasks are placed and from which tasks are taken for allocation to processors.
- We'll examine three algorithms:
  1. Manager/Worker
  2. Hierarchical Manager/Worker
  3. Decentralized Schemes

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.

Hierarchical Manager/Worker

- This variant divides workers into disjoint sets, each with a submanager.
- Workers request tasks from submanagers.
- Submanagers communicate periodically with the manager and with other submanagers to balance overall load.

Decentralized schemes

- No central manager.
- Separate task pool is maintained on each processor.
- Idle workers request problems from other processors.

Termination detection

- No matter which scheme is used, we need a mechanism for determining when a job is complete; otherwise, idle workers will never stop requesting work from other workers.
- 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.

Mapping design checklist

Questions to consider before finishing the Mapping step:

1. If considering an SPMD design for a complex problem, have you also considered an algorithm based on dynamic task creation and deletion?
2. If considering a design based on dynamic task creation and deletion, have you also considered an SPMD algorithm?
3. If using a centralized load-balancing scheme, have you verified that the manager will not become a bottleneck?
4. If using a dynamic load-balancing scheme, have you evaluated the relative costs of different strategies?
5. If using probabilistic or cyclic methods, do you have a large enough number of tasks to ensure reasonable load balance?