# Designing Parallel Algorithms

Zhengxiong Hou

Fall, 2022

# Designing Parallel Algorithms

- **Introduction of developing parallel codes**
- **Problem Analysis**
- **Parallel Algorithms Design Methodology:**

  **PCAM (Partitioning, Communications, Agglomeration, Mapping)**
- **Some more considerations:**
  - **Synchronization**
  - **Data Dependencies**
  - **I/O**

## A Generic Development Cycle (1)

- **Analysis**

  -Find the hotspot and understand its logic

- **Design**

  -Identify the concurrent tasks and their dependencies

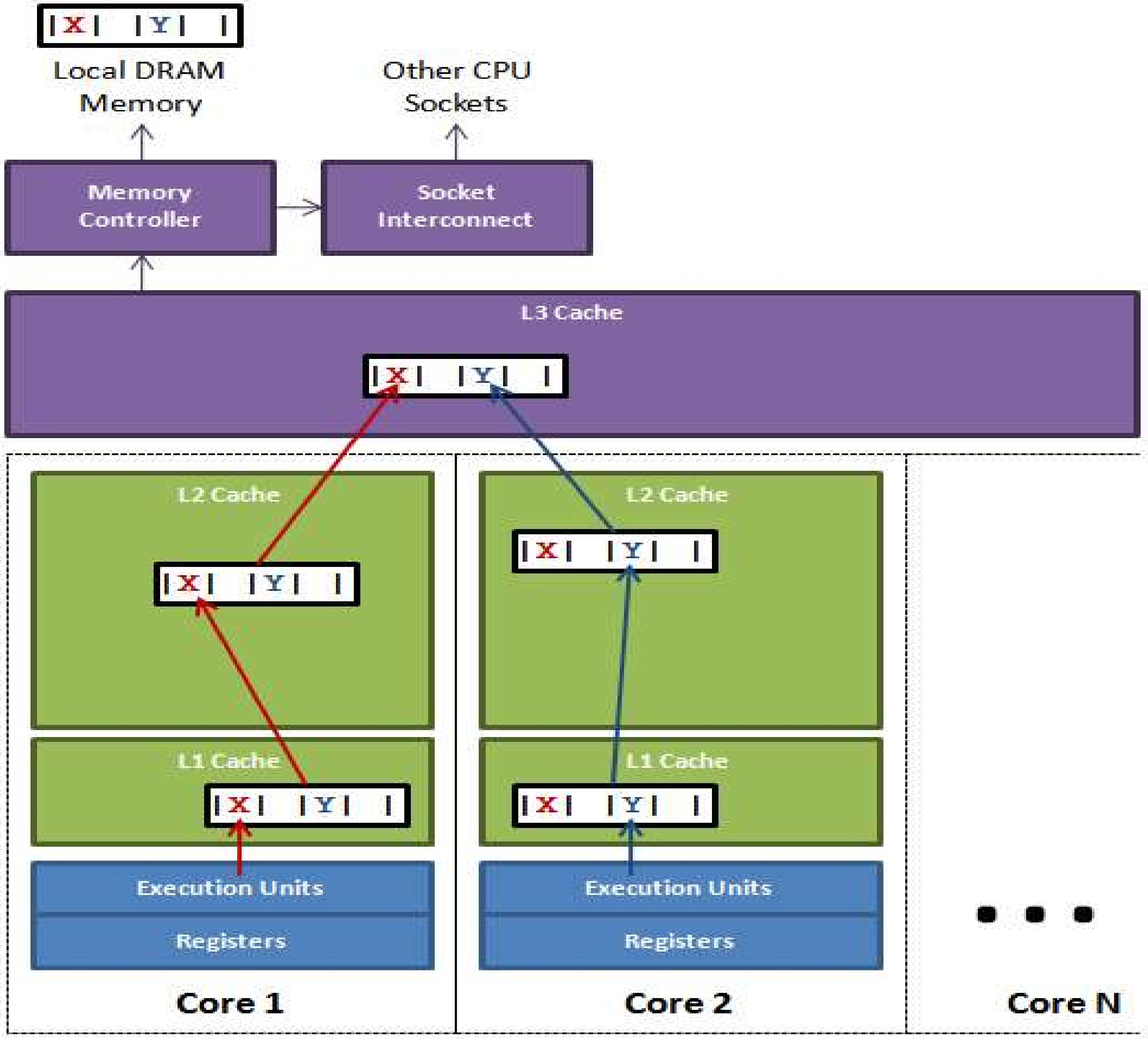
  -Decompose the whole dataset with minimal overhead of sharing or data movement between tasks

  -Introduce the proper parallel algorithm

  -Use proved parallel implementations

  -Memory management

  - Avoid heap contention among threads
  - Use thread-local storage to reduce synchronization
  - Detecting memory saturation in threaded applications
  - Avoid and identifying false sharing among threads

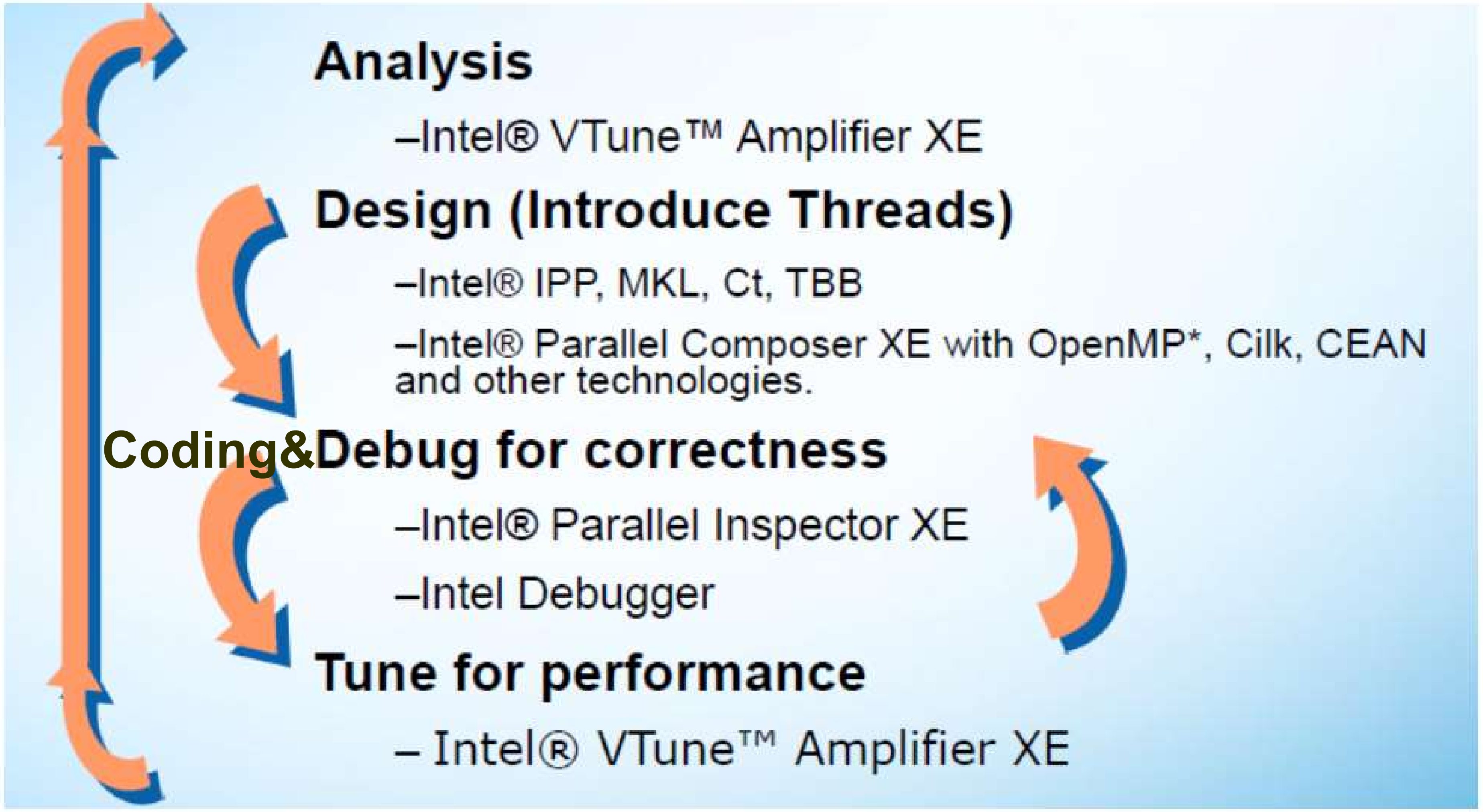


|X| |Y| |
Local DRAM Memory
Other CPU Sockets
Memory Controller
Socket Interconnect
L3 Cache
|X| |Y| |
L2 Cache
|X| |Y| |
L2 Cache
|X| |Y| |
L1 Cache
|X| |Y| |
L1 Cache
|X| |Y| |
Execution Units
Registers
Execution Units
Registers
Core 1
Core 2
• • •
Core N

## A Generic Development Cycle (2)

- Debug for correctness
  - Detect race conditions, deadlock, & memory issues
- Tune for performance
  - Balance the workload
  - Adjust lock & wait
  - Reduce thread operation overhead
  - Set the right granularity
  - Benchmark for scalibility

## Intel Generic Development Cycle



**Analysis**

–Intel® VTune™ Amplifier XE

**Design (Introduce Threads)**

–Intel® IPP, MKL, Ct, TBB

–Intel® Parallel Composer XE with OpenMP*, Cilk, CEAN and other technologies.

**Coding&Debug for correctness**

–Intel® Parallel Inspector XE

–Intel Debugger

**Tune for performance**

– Intel® VTune™ Amplifier XE

## Automatic vs. Manual Parallelization

- Designing and developing parallel algorithms and programs has characteristically been a very **manual process**. The programmer is typically responsible for both identifying and actually implementing parallelism.
- Very often, manually developing parallel codes is a time consuming, complex, error-prone and ***iterative*** process.
- For a number of years now, various tools have been available to **assist** the programmer with converting serial programs into parallel programs. The most common type of tool used to automatically parallelize a serial program is a **parallelizing compiler or pre-processor**.

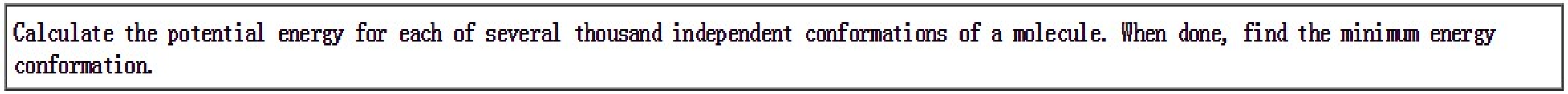
## A parallelizing compiler generally works in two different ways:

- Fully Automatic
  - The compiler analyzes the source code and identifies opportunities for parallelism.
  - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
  - **Loops (do, for)** are the most frequent target for automatic parallelization.
- Programmer Directed
  - Using "**compiler directives**" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
  - May be able to be used in conjunction with some degree of automatic parallelization also.

- The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP).
- If you are beginning with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer. However, there are several important caveats that apply to automatic parallelization: Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the compiler analysis suggests there are inhibitors or the code is too complex
- The following contents apply to  the **manual method** of developing parallel codes.

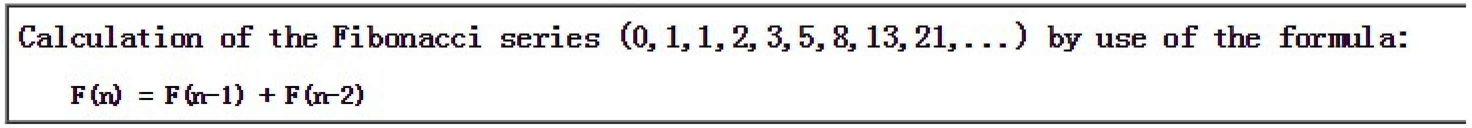
## Analysis-Understanding the Problem

- Undoubtedly, **the first step** in developing parallel software is to **understand the problem** that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.
  - Example of Parallelizable Problem:


Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

  - This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

## Example of a Non-parallelizable Problem:

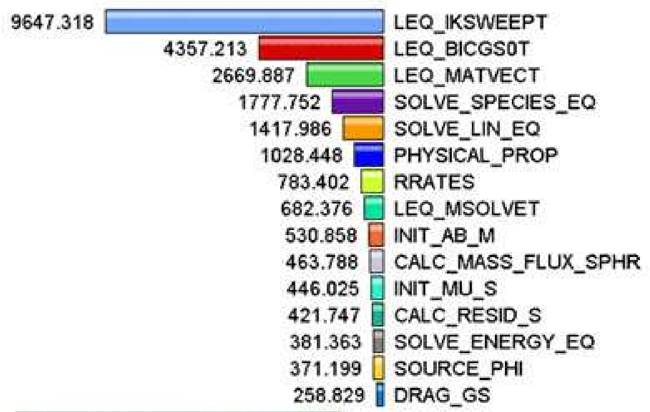

Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:

F(n) = F(n-1) + F(n-2)

- This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the F(n) value uses those of both F(n-1) and F(n-2). These three terms cannot be calculated independently and therefore, not in parallel.

## Identify the program's ***hotspots***:

- Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
- Profilers and performance analysis tools can help here
- Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.



- Identify ***bottlenecks*** in the program:
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
- Identify inhibitors to parallelism. One common class of inhibitor is ***data dependence***, as demonstrated by the Fibonacci sequence above.
- Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.
- Take advantage of optimized third party parallel software and highly optimized **math libraries** available from leading vendors (IBM's ESSL, Intel's MKL, AMD's AMCL, etc.).

## Methodical Design-PCAM (Foster)

- ***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.

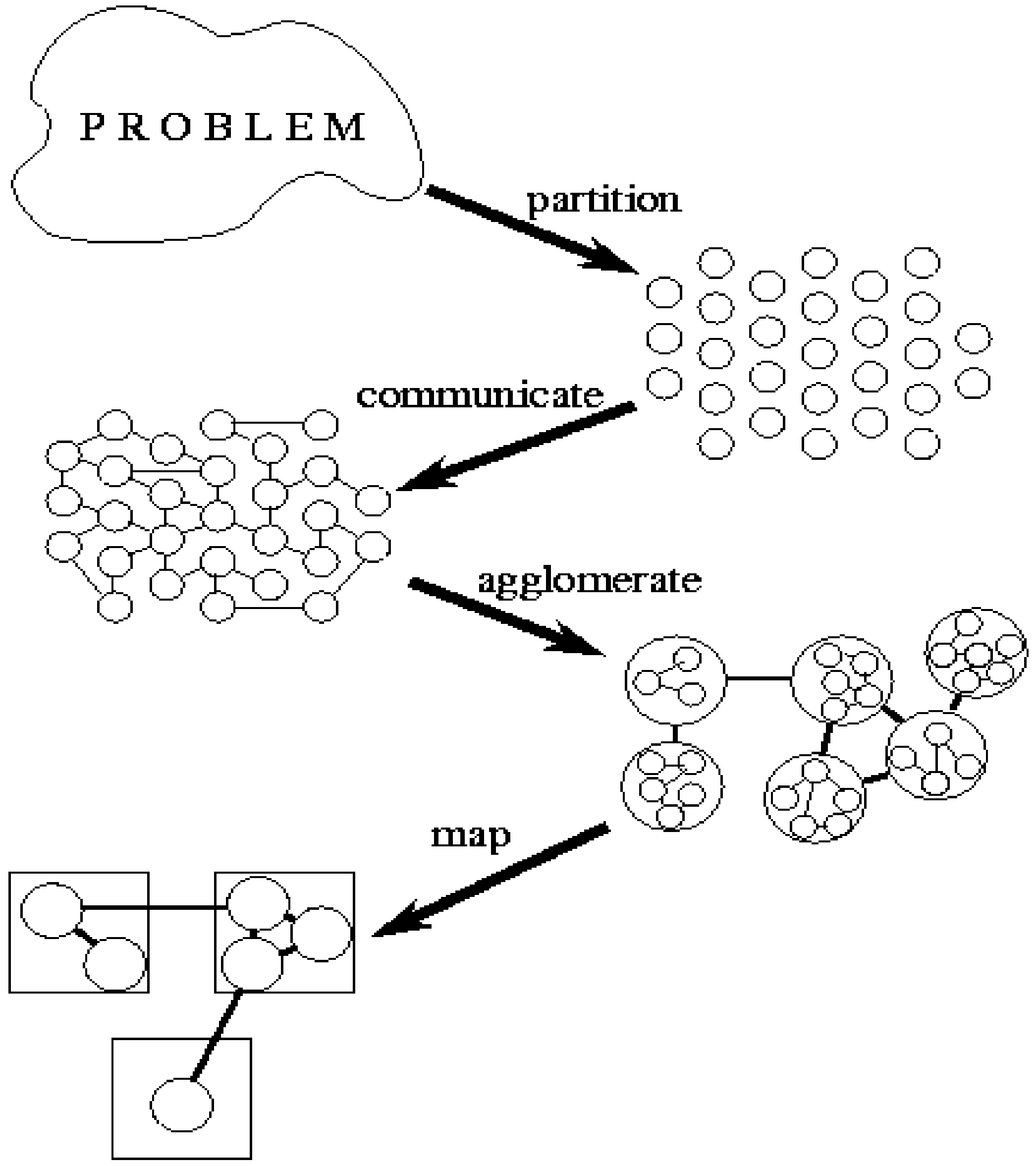
- ***Agglomeration.***

  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.

- *PCAM: a design methodology for parallel programs. Starting with a problem specification, we develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.*

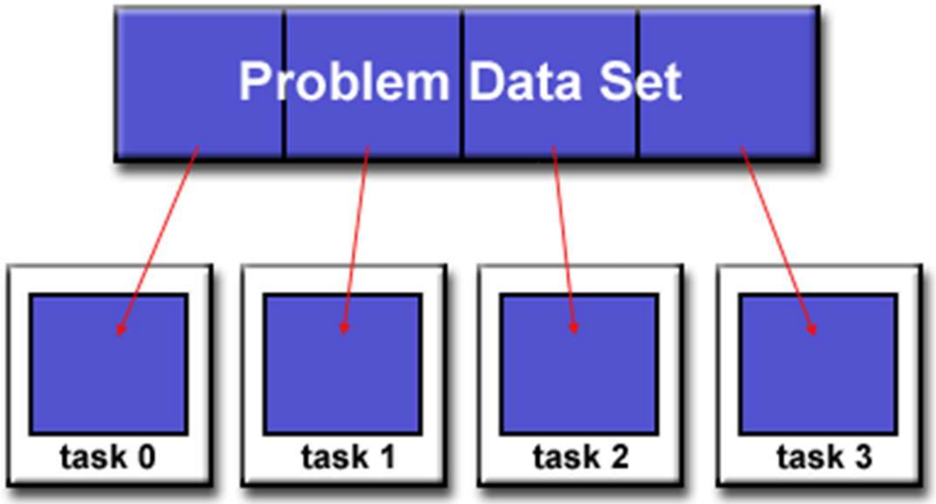


## Partitioning

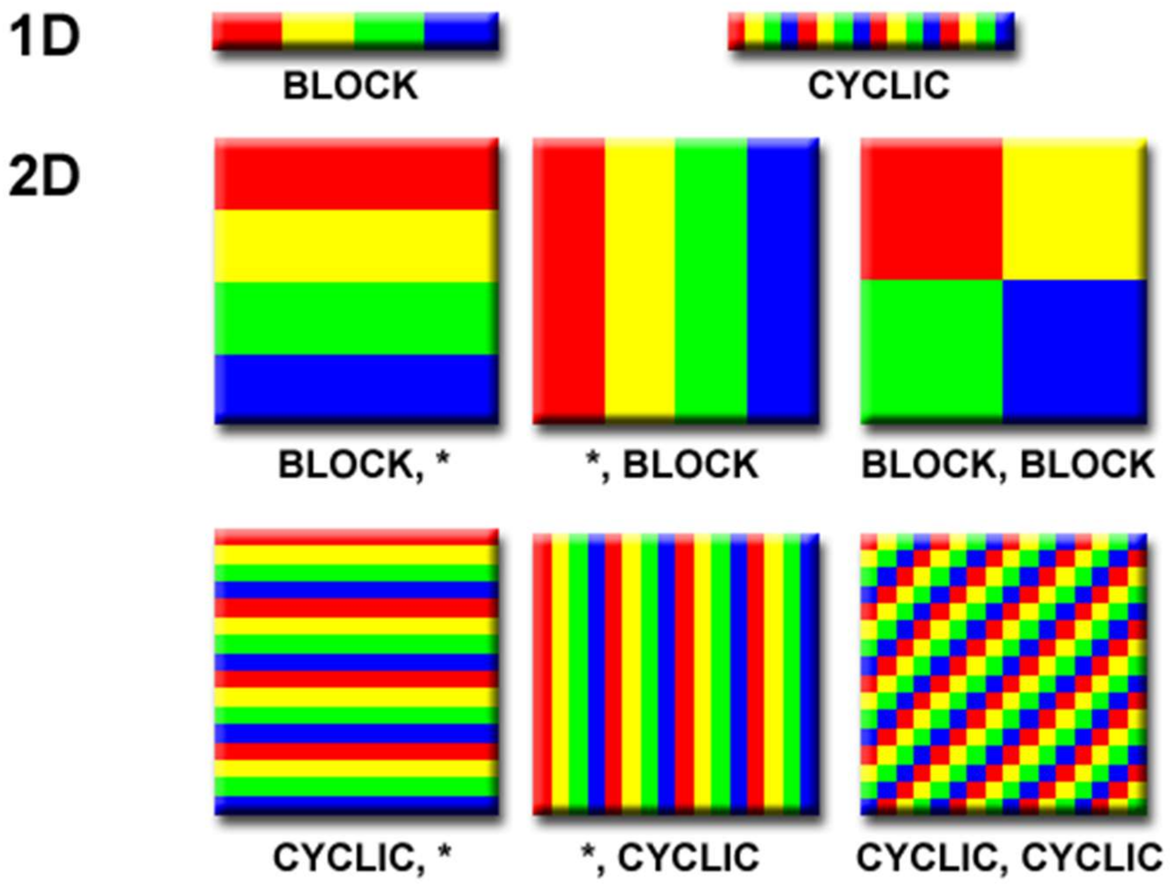
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.
- There are two basic ways to partition computational work among parallel tasks: ***domain decomposition*** and ***functional decomposition***.
- Combining these two types of problem decomposition is common and natural.

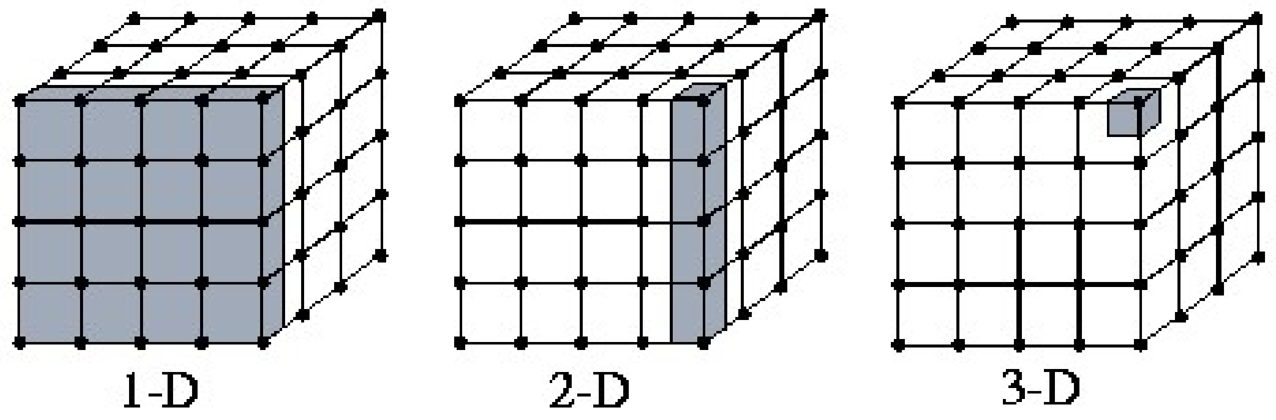
## Domain Decomposition:

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.



## There are different ways to partition data:





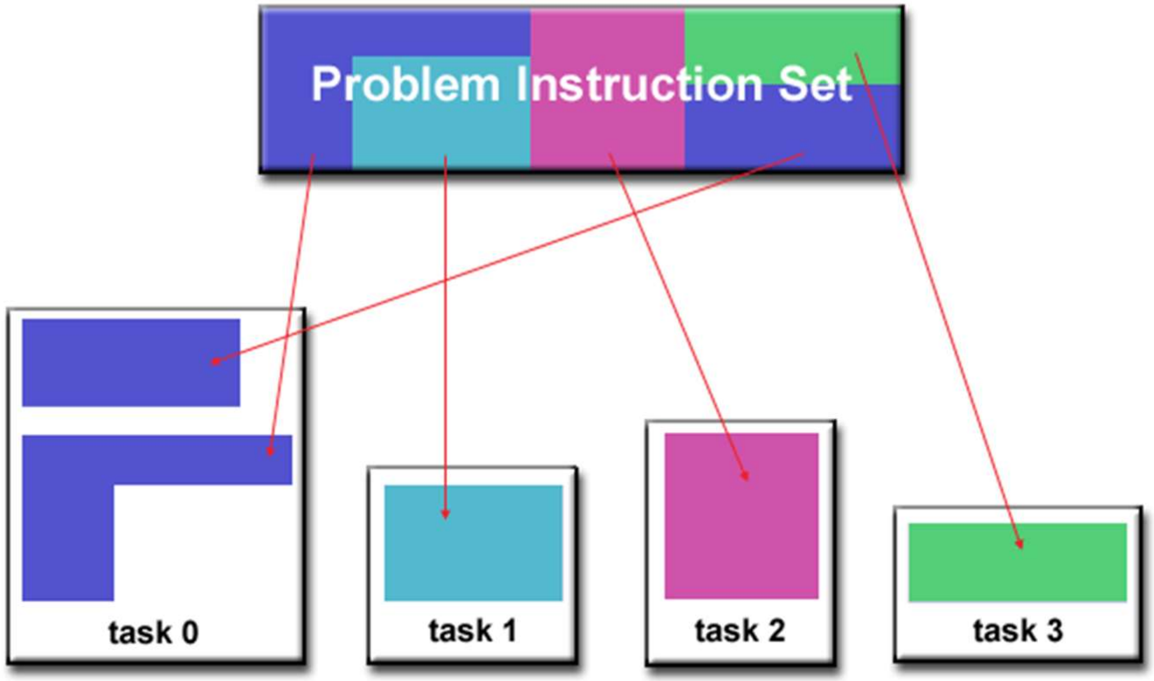
- *Domain decompositions for a problem involving a three-dimensional grid.* ***One-, two-, and three-dimensional decompositions*** *are possible;*

  *-in each case, data associated with a single task are shaded.*

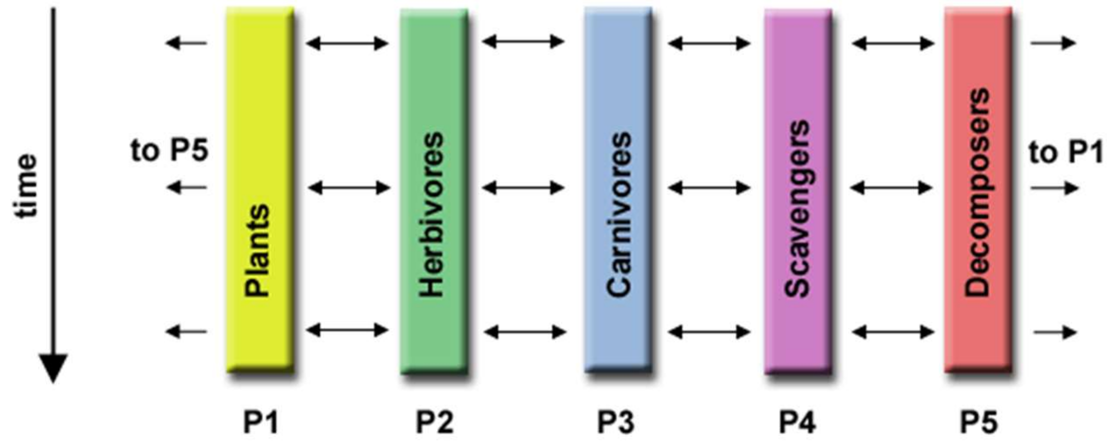
  *-A three-dimensional decomposition offers the greatest flexibility and is adopted in the early stages of a design.*

## Functional Decomposition:

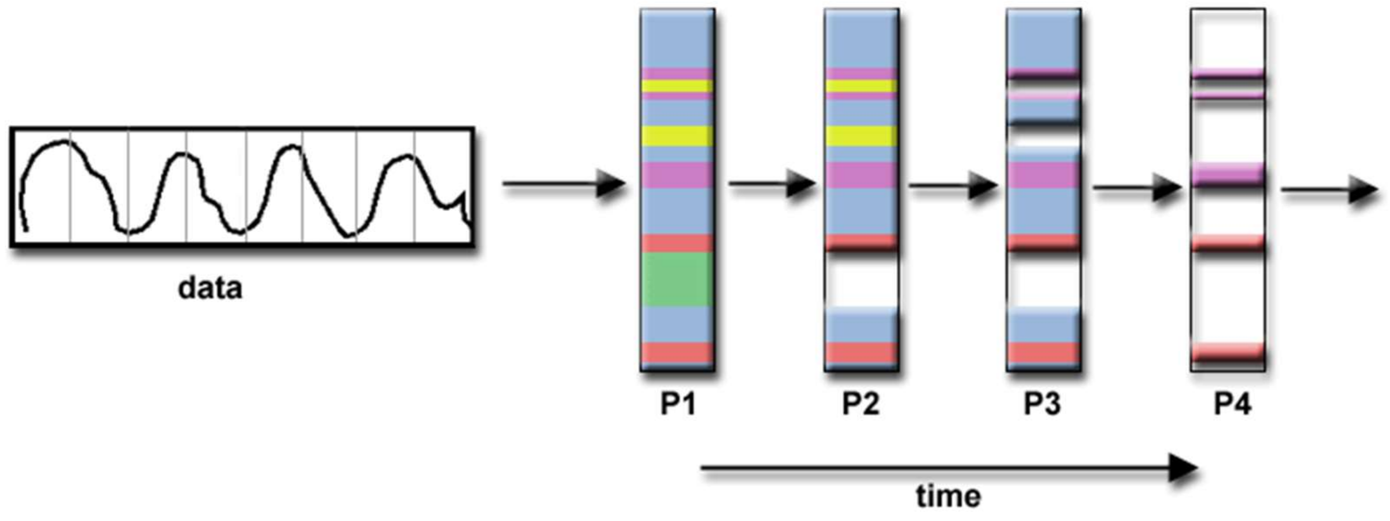
- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.



- Functional decomposition lends itself well to problems that can be split into different tasks.
  - **E.g. Ecosystem Modeling**
    Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.

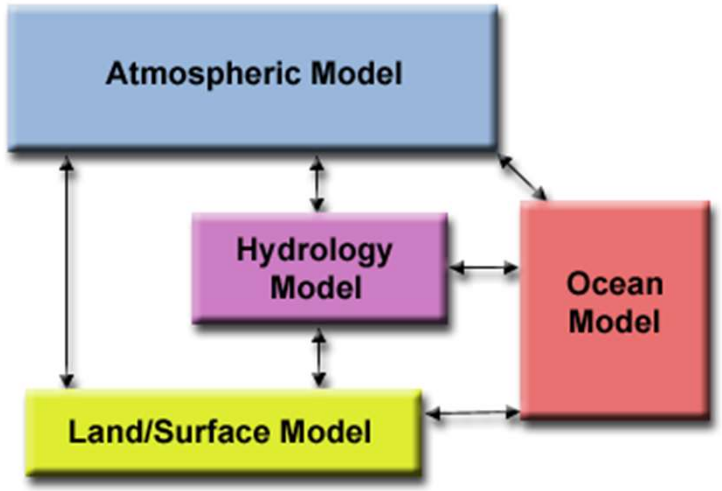


- **Signal Processing**
  - An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



## Climate Modeling

- Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the **atmosphere model** generates wind velocity data that are used by the ocean model, the **ocean model** generates sea surface temperature data that are used by the atmosphere model, and so on.



# Partitioning Design Checklist

- 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.
- Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
- 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. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
- Have you identified several alternative partitions? You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decompositions.

## Communications

- **Who Needs Communications?**
  - The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The need for communications between tasks depends upon your problem
- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
  - These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.
- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

## Factors to Consider:

- There are a number of important factors to consider when designing your program's inter-task communications:

- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

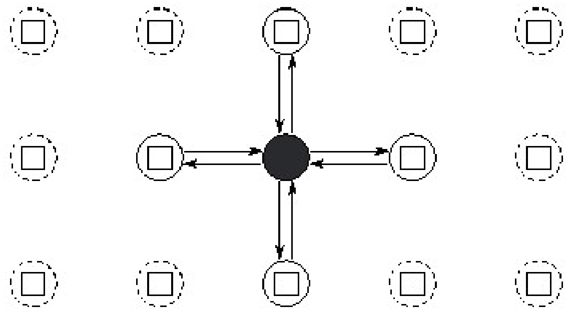
- **Latency vs. Bandwidth**
  - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
  - ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

- **Visibility of communications**
  - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
  - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.

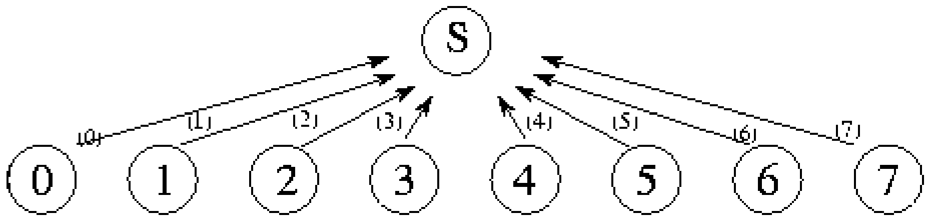
## Communication patterns

- For clarity in exposition, we categorize communication patterns along four loosely orthogonal axes:
  - **-local/global,**
  - **-structured/unstructured,**
  - **-static/dynamic,**
  - **-synchronous/asynchronous.**

- In ***local* communication**, each task communicates with a small set of other tasks (its ``neighbors");

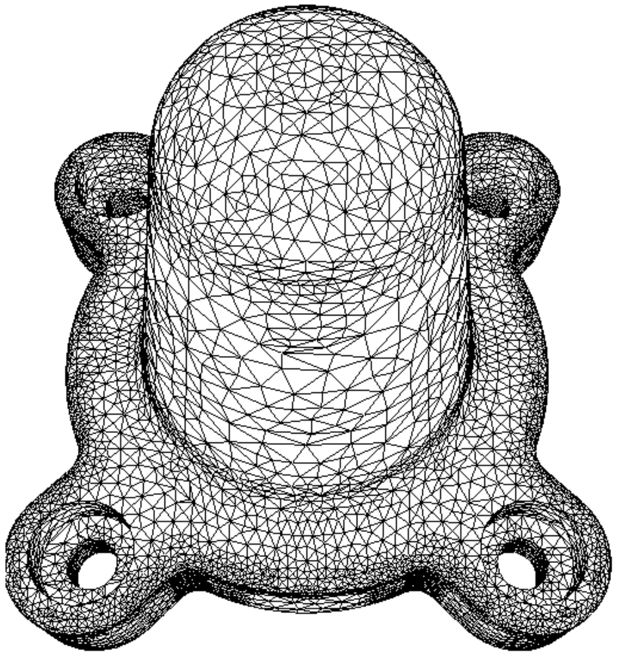


- in contrast, ***global* communication** requires each task to communicate with many tasks.



- In ***structured* communication**, a task and its neighbors form a regular structure, such as a tree or grid;

  in contrast, ***unstructured* communication** networks may be arbitrary graphs.



*-Example of a problem requiring unstructured communication. In this finite element mesh generated for an assembly part, each vertex is a grid point. An edge connecting two vertices represents a data dependency that will require communication if the vertices are located in different tasks. Notice that different vertices have varying numbers of neighbors. (Image courtesy of M. S. Shephard.)*

- In ***static* communication**, the identity of communication partners does not change over time;

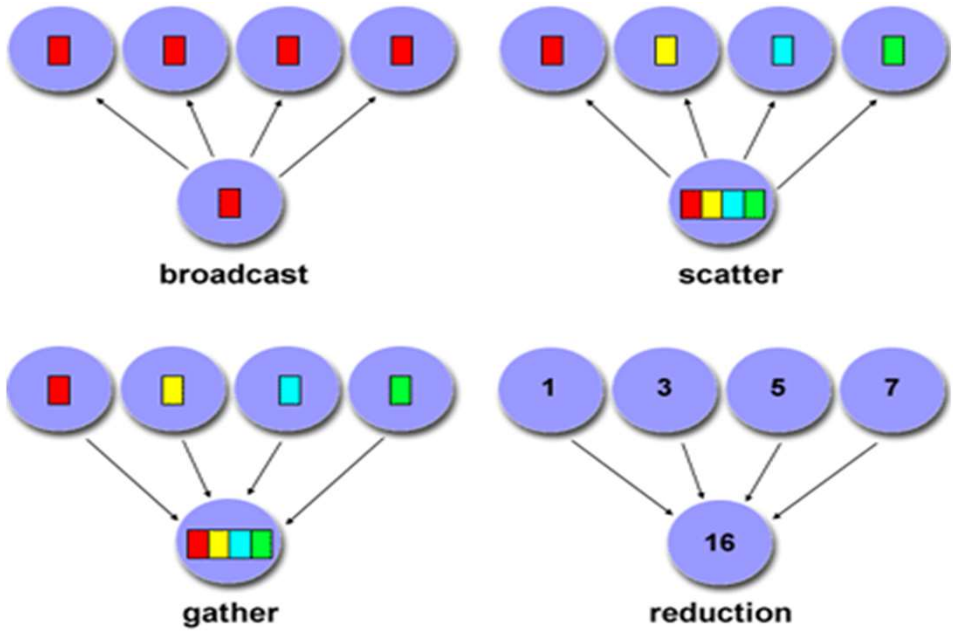
  -in contrast, the identity of communication partners in ***dynamic* communication** structures may be determined by data computed at runtime and may be highly variable.

- In ***synchronous* communication**, producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations;

  -in contrast, ***asynchronous* communication** may require that a consumer obtain data without the cooperation of the producer.

- **Synchronous vs. asynchronous communications**
  - **Synchronous communications** require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to ***blocking*** communications since other work must wait until the communications have completed.
  - **Asynchronous communications** allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
  - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
  - **Interleaving computation with communication** is the single greatest benefit for using asynchronous communications.

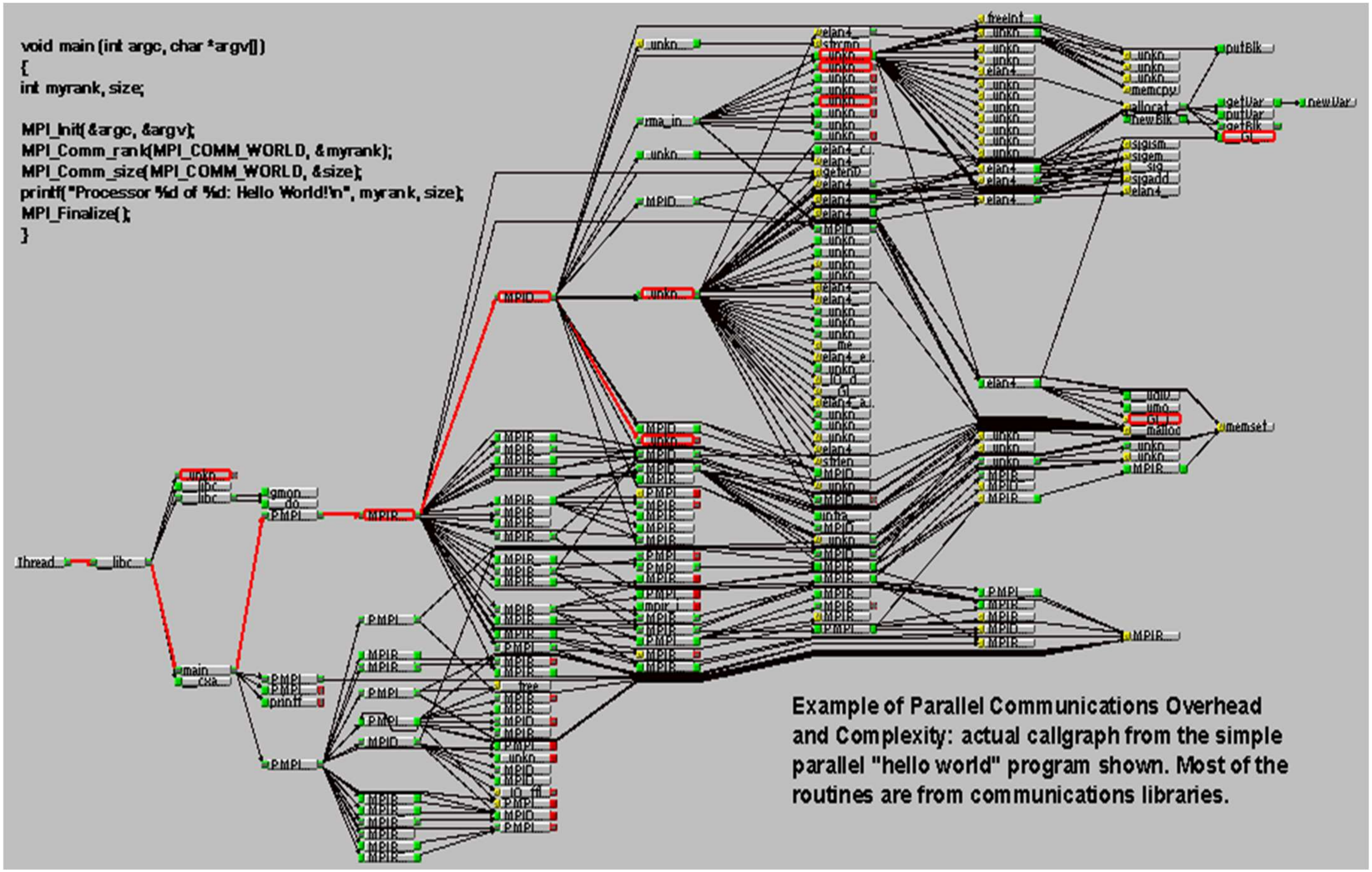
- **Scope of communications**
  - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
  - **Point-to-point** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - **Collective** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more):



## Efficiency of communications

- Very often, the programmer will have a choice with regard to factors that can affect communications performance. Only a few are mentioned here.
  - **Which implementation** for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
  - **What type of communication operations** should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
  - **Network media** - some platforms may offer more than one network for communications. Which one is best?

# Overhead and Complexity



```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init( &argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf( "Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize( );
}
```

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

## Communication Design Checklist

- **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.

- **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

- **Are communication operations able to proceed concurrently?**

  -If not, your algorithm is likely to be inefficient and nonscalable. Try to use divide-and-conquer techniques to uncover concurrency

- **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. You may also wish to revisit your problem specification

## Agglomeration

- We revisit 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
- In particular, we consider 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. We also determine whether it is worthwhile to *replicate* data and/or computation.

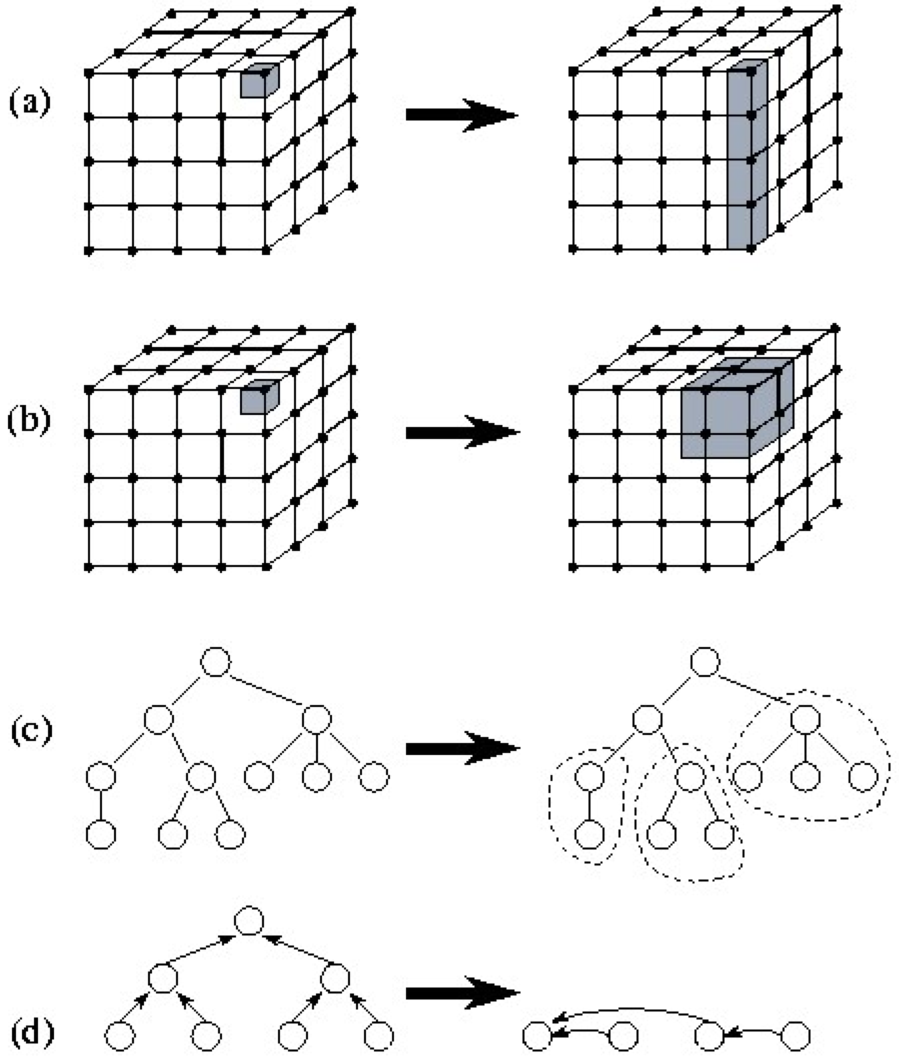
- ***Examples of agglomeration***

  *In (a), the size of tasks is increased by reducing the dimension of the decomposition from three to two.*

  *In (b), adjacent tasks are combined to yield a three-dimensional decomposition of higher granularity.*

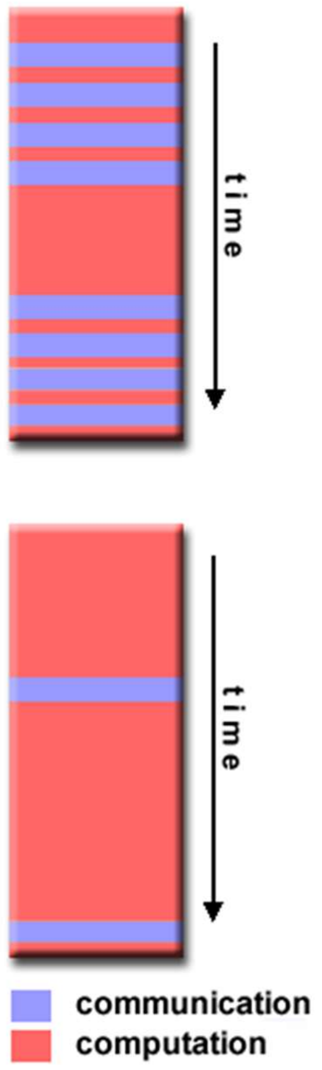
  *In (c), subtrees in a divide-and-conquer structure are coalesced.*

  *In (d), nodes in a tree algorithm are combined.*



## Granularity

- **Computation / Communication Ratio:**
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - Periods of computation are typically separated from periods of communication by synchronization events.



## Fine-grain Parallelism:

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio

+Facilitates load balancing

-Implies high communication overhead and less opportunity for performance enhancement

-If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

## Coarse-grain Parallelism:

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio

+Implies more opportunity for performance increase

-Harder to load balance efficiently

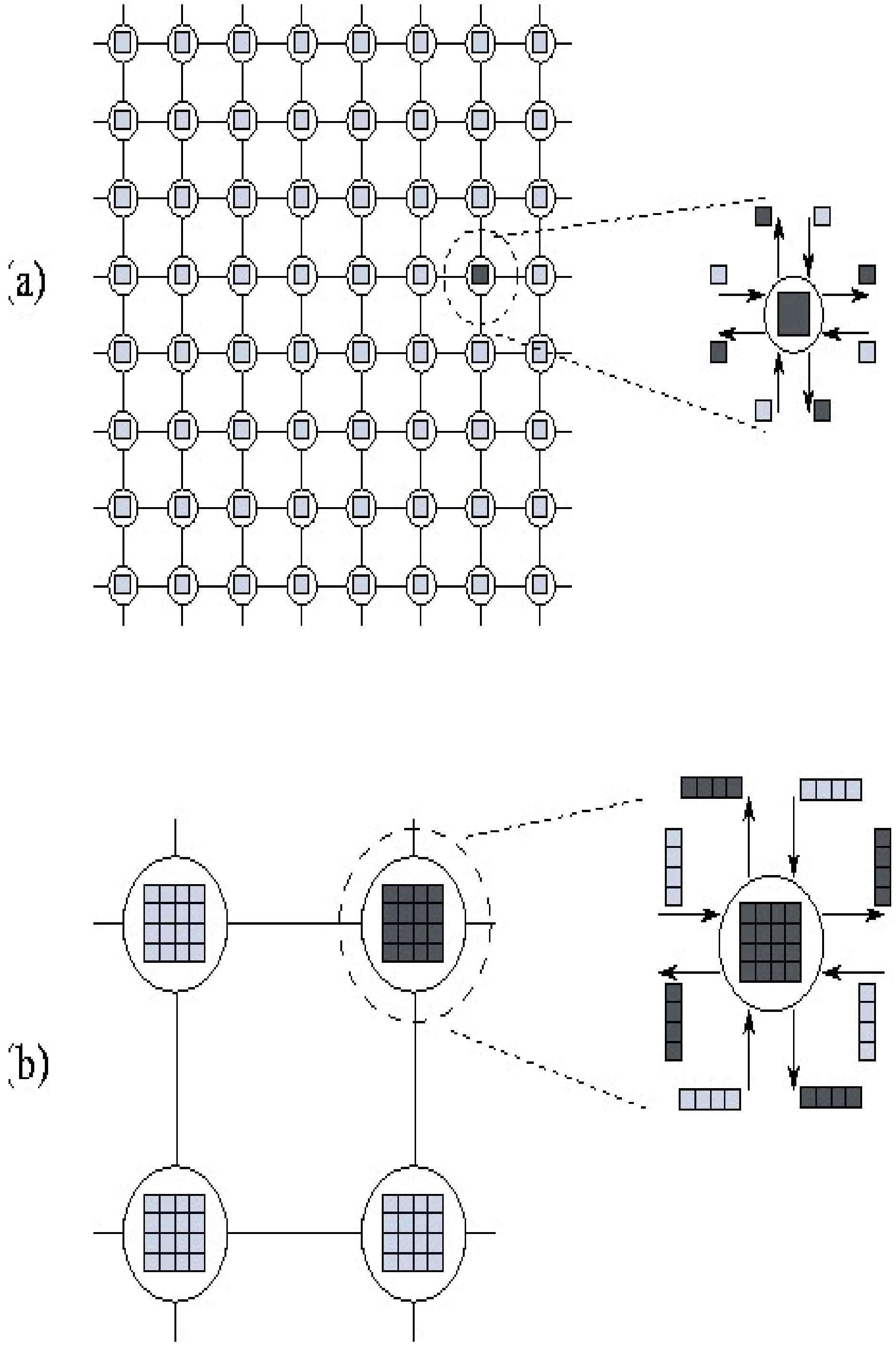
## Which is Best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

## An Example of Increasing Granularity

- *Effect of increased granularity on communication costs in a two-dimensional finite difference problem with a five-point stencil. The figure shows **fine- and coarse-grained two-dimensional partitions** of this problem.*

  *In each case, a single task is exploded to show its outgoing messages (dark shading) and incoming messages (light shading).*
- *In (a), a computation on an $8 \times 8$ grid is partitioned into 64 tasks, each responsible for a single point, while in (b) the same computation is partitioned into 4 tasks, each responsible for 16 points.*
- *In (a), 256 communications are required, 4 per task; these transfer a total of 256 data values. In (b), only 16 communications are required, and only 64 data values are transferred.*



## Agglomeration Design Checklist

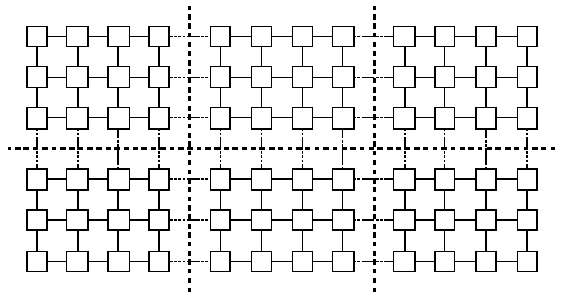
- We have now revised the partitioning and communication decisions developed in the first two design stages by agglomerating tasks and communication operations.
- We may have agglomerated tasks because analysis of communication requirements shows that the original partition created tasks that cannot execute concurrently. Alternatively, we may have used agglomeration to increase computation and communication granularity and/or to decrease software engineering costs, even though opportunities for concurrent execution are reduced.
- At this stage, we evaluate our design with respect to the following checklist.

- **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.
- 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?
- 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?
- Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created by agglomeration, 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.

- 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.
- 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; performance models can be used to quantify these tradeoffs.
- Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability? Other things being equal, algorithms that create fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks.
- If you are parallelizing an existing sequential program, have you considered the cost of the modifications required to the sequential code? If these costs are high, consider alternative agglomeration strategies that increase opportunities for code reuse. If the resulting algorithms are less efficient, use performance modeling techniques to estimate cost tradeoffs.

## Mapping

- In this stage of the parallel algorithm design process, we specify where each task is to execute.
- This 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**. In general, mapping remains a difficult problem that must be explicitly addressed when designing parallel algorithms.



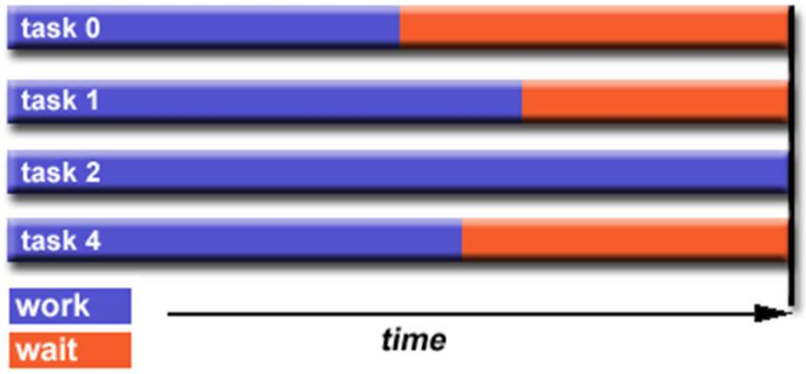
- Our basic goal in developing mapping algorithms is normally to **minimize total execution time**. We use two strategies to achieve this goal:

  -We place tasks that are able to execute concurrently on *different* processors, so as to **enhance concurrency**.

  -We place tasks that communicate frequently on the *same* processor, so as to **increase locality**.

## Load Balancing

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



## How to Achieve Load Balance:

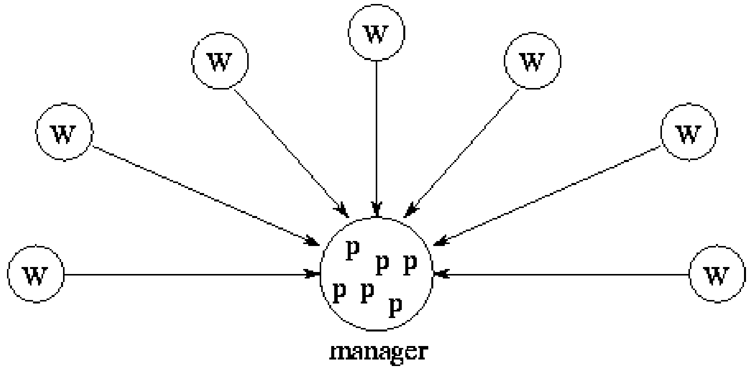
- **Equally partition the work each task receives**
  - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

## Use dynamic work assignment

- Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
  - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
  - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
  - *N*-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
- When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
- It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

## 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 discuss **manager/worker, hierarchical manager/worker,** and **decentralized** approaches.



## Mapping Design Checklist

- 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.
- 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.
- 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 the tasks themselves, to the manager.

- If using a **dynamic load-balancing** scheme, have you evaluated the relative costs of different strategies? Be sure to include the implementation costs in your analysis. Probabilistic or cyclic mapping schemes are simple and should always be considered, because they can avoid the need for repeated load-balancing operations.
- If using **probabilistic or cyclic methods**, do you have a large enough number of tasks to ensure reasonable **load balance**? Typically, at least ten times as many tasks as processors are required.

## Some more considerations:

- Synchronization
- Data Dependencies
- I/O

## Synchronization

- Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.
- Can be a significant factor in program performance (or lack of it)
- Often requires "serialization" of segments of the program.

## Types of Synchronization:

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.
- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking

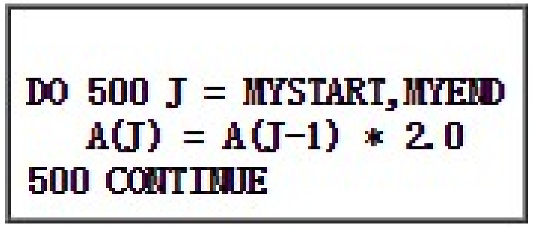
- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.
  - Discussed previously in the Communications section.

## Data Dependencies

- **Definition:**
  - A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
  - A data dependence results from multiple use of the same location(s) in storage by different tasks.
  - Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

## Examples:

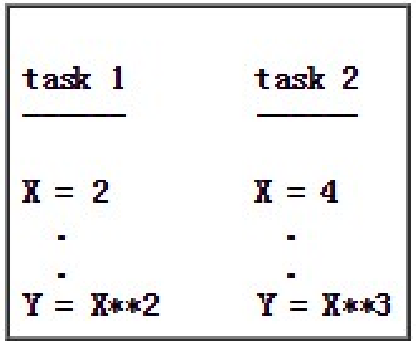
- **Loop carried data dependence**



```
DO 500 J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
500 CONTINUE
```

  - The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
- If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  - Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  - Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

- **Loop independent data dependence**



```
task 1        task 2
------        ------

X = 2         X = 4
  .             .
  .             .
Y = X**2      Y = X**3
```

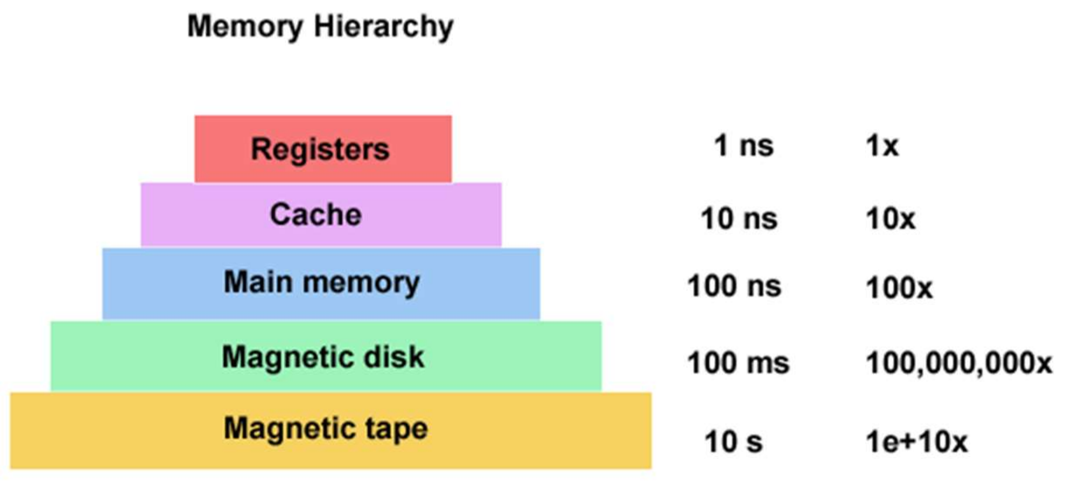
- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture - if or when the value of X is communicated between the tasks.
  - Shared memory architecture - which task last stores the value of X.
- Although all data dependencies are important to identify when designing parallel programs, **loop carried dependencies** are particularly important since loops are possibly the most common target of parallelization efforts.

## How to Handle Data Dependencies:

- Distributed memory architectures - communicate required data at synchronization points.
- Shared memory architectures -synchronize read/write operations between tasks.

# I/O

- **The Bad News:**
  - I/O operations are generally regarded as inhibitors to parallelism.
  - I/O operations require orders of magnitude more time than memory operations.
  - Parallel I/O systems may be immature or not available for all platforms.
  - In an environment where all tasks see the same file space, write operations can result in file overwriting.
  - Read operations can be affected by the file server's ability to handle multiple read requests at the same time.
  - I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers.



## The Good News:

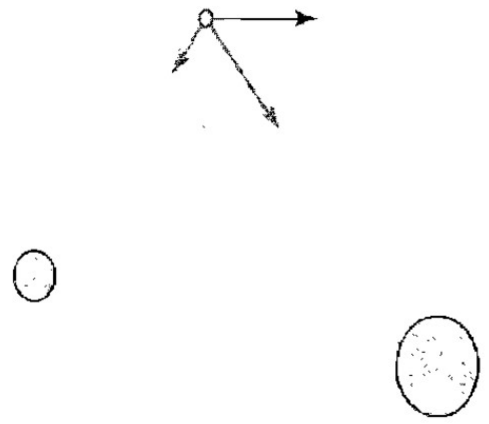
- Parallel file systems are available. For example:
  - GPFS: General Parallel File System (IBM)
  - Lustre: for Linux clusters (Intel)
  - OrangeFS: Open source parallel file system follow on to Parallel Virtual File System (PVFS)
  - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
  - And more - see http://en.wikipedia.org/wiki/List_of_file_systems#Distributed_parallel_file_systems
- The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

- **A few pointers:**
  - Rule #1: Reduce overall I/O as much as possible
  - If you have access to a parallel file system, use it.
  - Writing large chunks of data rather than small chunks is usually significantly more efficient.
  - Fewer, larger files performs better than many small files.
  - Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
  - Aggregate I/O operations across tasks - rather than having many tasks perform I/O, have a subset of tasks perform it.

## Case study of designing parallel algorithm

## Case1: The n-body problem (molecular dynamics)

- In a Newtonian n-body simulation, gravitational forces have infinite range
- We consider the parallelization of a sequential algorithm in which a computation is performed on every pair of objects
- We are simulating the motion of **n particles** of varying mass in two dimensions. During each iteration of the algorithm we need to compute the new position and velocity vector of each particle, given the positions of all the other particles.

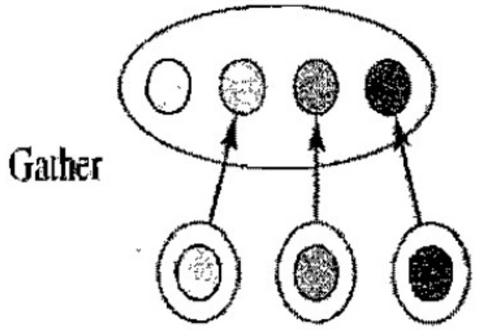


## Partitioning

- Our first step is to partition the dataset. To start with, let's assume **we have one task per particle**.

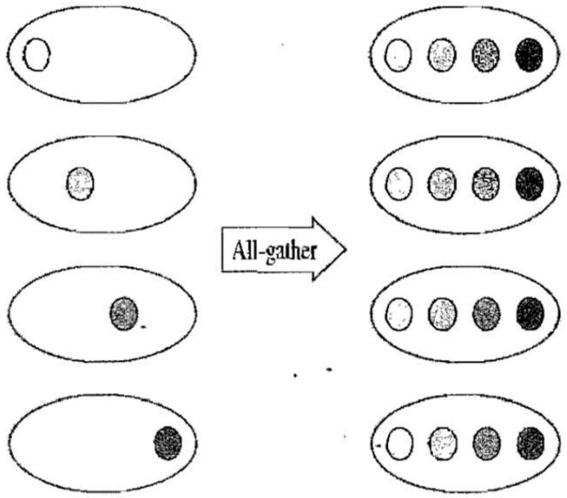
- In order for this task to compute the new location of the particle, it must know the locations of all the other particles

## Communication

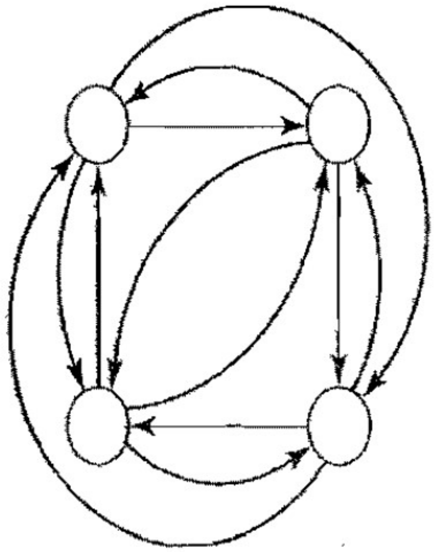


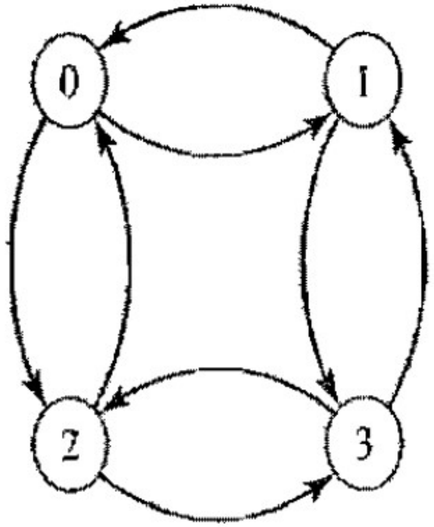
- A **gather** operation is a global communication that takes a dataset distributed among a group of tasks and collects the items on a single task

- An **all-gather** operation is similar to gather, except at the end of the communication every task has a copy of the entire dataset



- In this case, we want to update the location of every particle, so an all-gather communication is called for.





- ✓ One way to make all data values available to all tasks is to set up a channel between every pair of tasks.
- ✓ The all-gather data communication requires that each task have only log $p$ outgoing channels and log $p$ incoming channels.

## Agglomeration and Mapping

- In general, there are far more particles n than processors *p.* Let's assume that *n* is a multiple of *p.* We associate one task per processor and agglomerate *n/p* particles into each task.
- Now the all-gather communication operation requires log *p* communication steps. In the first step the messages have length *n/p,* in the second step the messages have length 2n/*p,* etc.

# Case 2-Atmosphere Model

- **Atmosphere Model Background**

-An *atmosphere model* is a computer program that simulates atmospheric processes (wind, clouds, precipitation, etc.) that influence weather or climate. It may be used to study the evolution of tornadoes, to predict tomorrow's weather, or to study the impact on climate of increased concentrations of atmospheric carbon dioxide.

-Like many numerical models of physical processes, an atmosphere model solves a set of **partial differential equations**, in this case describing the basic fluid dynamical behavior of the atmosphere

Conservation of momentum:

$$\frac{du}{dt} - \left(f + u\frac{\tan\phi}{a}\right)v = -\frac{1}{a\cos\phi}\frac{1}{\rho}\frac{\partial p}{\partial \lambda} + F_\lambda$$

$$\frac{dv}{dt} + \left(f + u\frac{\tan\phi}{a}\right)u = -\frac{1}{\rho a}\frac{\partial p}{\partial \phi} + F_\phi$$

Hydrostatic approximation:

$$g = -\frac{1}{\rho}\frac{\partial p}{\partial z}$$

Conservation of mass:

$$\frac{\partial \rho}{\partial t} = -\frac{1}{a\cos\phi}\left(\frac{\partial}{\partial \lambda}(\rho u) + \frac{\partial}{\partial \phi}(\rho v\cos\phi)\right) - \frac{\partial}{\partial z}(\rho w)$$

Conservation of energy:

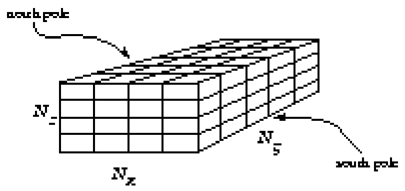
$$C_p\frac{dT}{dt} - \frac{1}{\rho}\frac{dp}{dt} = Q$$

State equation (atmosphere):

$$p = \rho RT$$

**Figure 2.20:** The basic predictive equations used in atmospheric modeling, where $\phi$ and $\lambda$ are latitude and longitude, $z$ is height, $u$ and $v$ are horizontal components of velocity, $p$ is pressure, $\rho$ is density, $T$ is temperature, $f$ is Coriolis force, $g$ is gravity, $F$ and $Q$ are external forcing terms, $C_p$ is specific heat, and $a$ is the earth's radius.

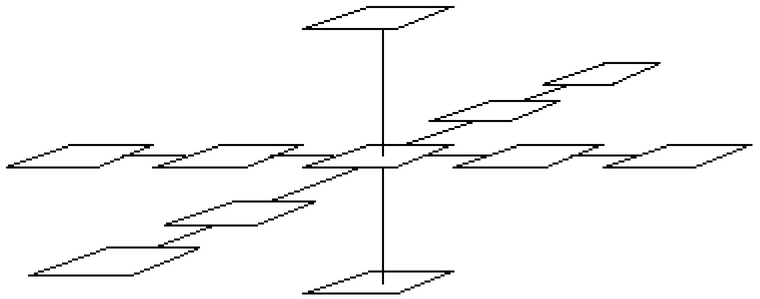
- The behavior of these equations on a continuous space is approximated by their behavior on a finite set of regularly spaced points in that space.



Typically, these points are located on a rectangular latitude-longitude grid of size $N_x \times N_y \times N_z$, with $N_z$ in the range 15--30, $N_x \approx 2N_y$, and $N_y$ in the range 50--500

This grid is periodic in the $x$ and $y$ dimensions, meaning that grid point $G_{0,0,0}$ is viewed as being adjacent to $G_{0,N_y-1,0}$ and $G_{N_x-1,0,0}$. A vector of values is maintained at each grid point, representing quantities as pressure, temperature, wind velocity, and humidity.

- The atmosphere model performs a **time integration** to determine the state of the atmosphere at some future time, based on an initial state. This integration proceeds in a series of steps, with each step advancing the state of the computation by a fixed amount.
- We shall assume that the model uses a **finite difference method** to update grid values, with **a nine-point stencil** being used to compute atmospheric motion in the horizontal dimension, and a **three-point stencil** in the vertical



- The following plate illustrates one of the many phenomena that can be simulated using an atmospheric circulation model.
- This shows a potential temperature isosurface of two thunderstorm downdrafts that hit the ground as microbursts, then spread out and collide. The surfaces outline the boundaries of the cold downdrafted air. The collision region contains wind fields that are dangerous to landing aircraft. The grey tiles are 1-kilometer squares and the model domain is $16 \times 18$ km with 50 m resolution.

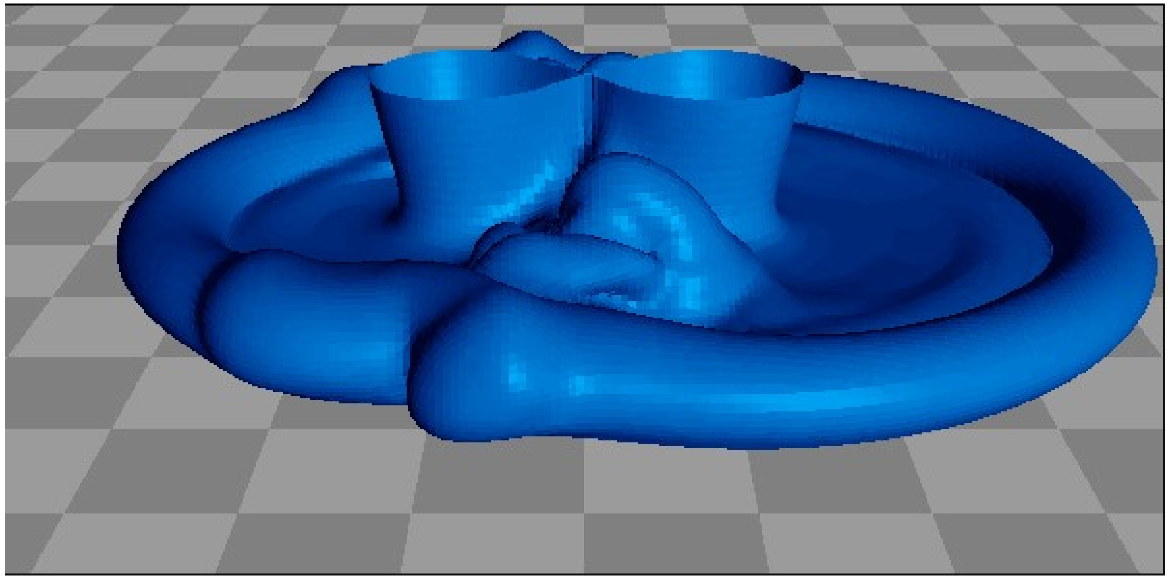


Plate : Potential temperature isosurface of two colliding thunderstorm microbursts. Image courtesy of J. Anderson.

## Atmosphere Model Algorithm Design

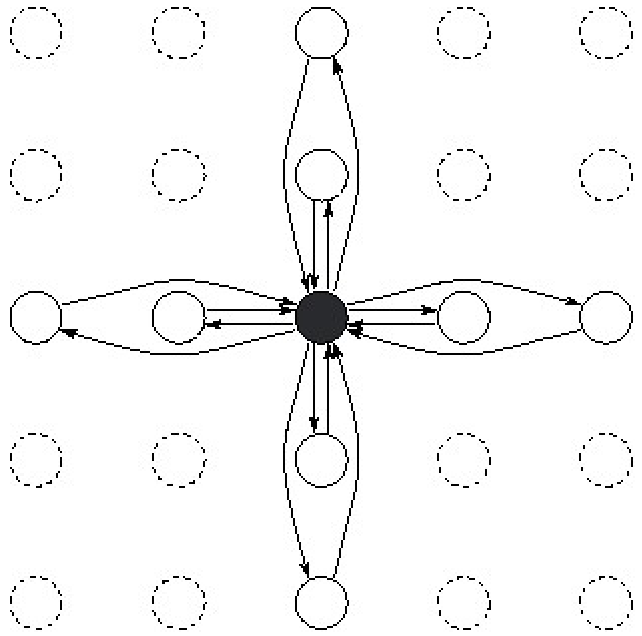
- **(1) Partition**

  The grid used to represent state in the atmosphere model is a natural candidate for domain decomposition. Decompositions in the *x* , *y* , and/or *z* dimensions are possible.

  Pursuant to our strategy of exposing the maximum concurrency possible, we initially favor the most aggressive decomposition possible, which in this case defines a task for each grid point. This task maintains as its state the various values associated with its grid point and is responsible for the computation required to update that state at each time step.

Hence, we have a total of $N_x \times N_y \times N_z$ tasks, each with $O(1)$ data and computation per time step.

## (2) Atmosphere Model Algorithm-Communication



- *task and channel structure for a two-dimensional finite difference computation with nine-point stencil, assuming one grid point per processor. Only the channels used by the shaded task are shown.*

## We identify three distinct communications:

- ***Finite difference stencils.***

  -If we assume a fine-grained decomposition in which each task encapsulates a single grid point, **the nine-point stencil used in the horizontal dimension** requires that each task obtain values from eight neighboring tasks. The corresponding channel structure is illustrated in previous figure.

  -Similarly, the **three-point stencil used in the vertical dimension** requires that each task obtain values from two neighbors.

- ***Global operations.***

The atmosphere model computes periodically the total mass of the atmosphere, in order to verify that the simulation is proceeding correctly. This quantity is defined as follows:

$$Total\ Mass = \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} \sum_{k=0}^{N_z-1} M_{ijk},$$

where $M_{ijk}$ denotes the mass at grid point *(i,j,k)*.

- ***Physics computations.***

  If each task encapsulates a single grid point, then the physics component of the atmosphere model requires considerable communication. For example, the total clear sky (TCS) at level $k \geq 1$ is defined as

$$\begin{aligned} \mathrm{TCS}_k &= \prod_{i=1}^{k} (1 - \mathrm{cld}_i) TCS_1 \\ &= TCS_{k-1}(1 - \mathrm{cld}_k), \end{aligned}$$

where level 0 is the top of the atmosphere and $\mathrm{cld}_i$ is the cloud fraction at level $i$.

This *prefix product* operation can be performed in $\log N_z$ steps using a variant of the hypercube algorithm

- In total, the physics component of the model requires on the order of 30 communications per grid point and per time step.

## Optimization of communication

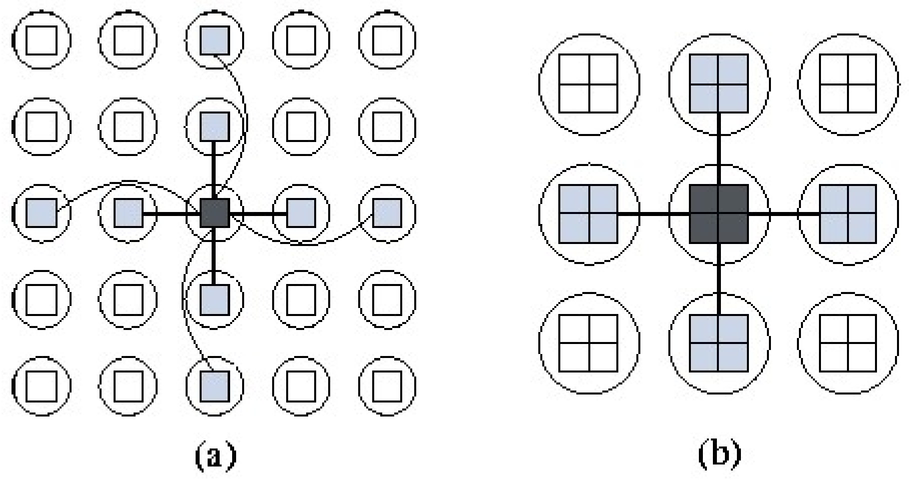
- The communication associated with the finite difference stencil is distributed and hence can proceed concurrently. So is the communication required for the global communication operation, thanks to the distributed algorithm

- The one component of our algorithm's communication structure that is problematic is the physics. However, we shall see that the need for this communication can be avoided by agglomeration.

## (3) Atmosphere Model Algorithm-Agglomeration

- Our fine-grained domain decomposition of the atmosphere model has created $N_x \times N_y \times N_z$ tasks: between $10^5$ and $10^7$ , depending on problem size. This is likely to be many more than we require and some degree of agglomeration can be considered.
- We identify three reasons for pursuing agglomeration:

## (a) Reduce the communication requirements



- *Using agglomeration to reduce communication requirements in the atmosphere model.*

  *-In (a), each task handles a single point and hence must obtain data from **eight other tasks** in order to implement the nine-point stencil.*

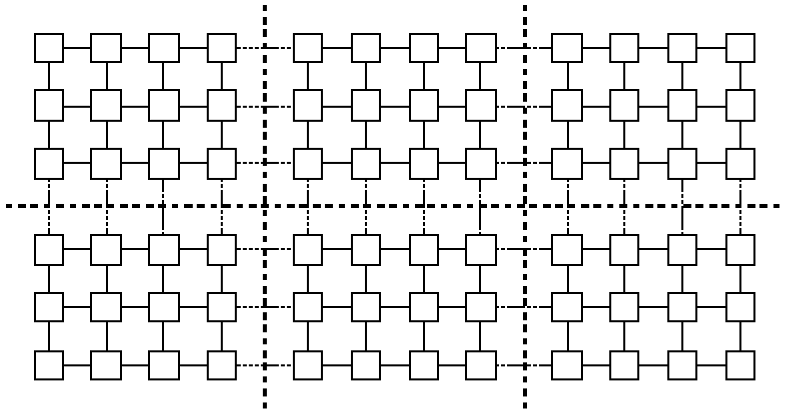
  *-In (b), granularity is increased to $2 \times 2$ points, meaning that only **4 communications** are required per task.*

## (b) Vertical dimension

- Communication requirements in the horizontal dimension are relatively small: a total of four messages containing eight data values.
- In contrast, the vertical dimension requires communication not only for the finite difference stencil (2 messages, 2 data values) but also for various ``physics'' computations (30 messages). These communications can be avoided by agglomerating tasks within each vertical column.

## (4) Atmosphere Model Algorithm-Mapping

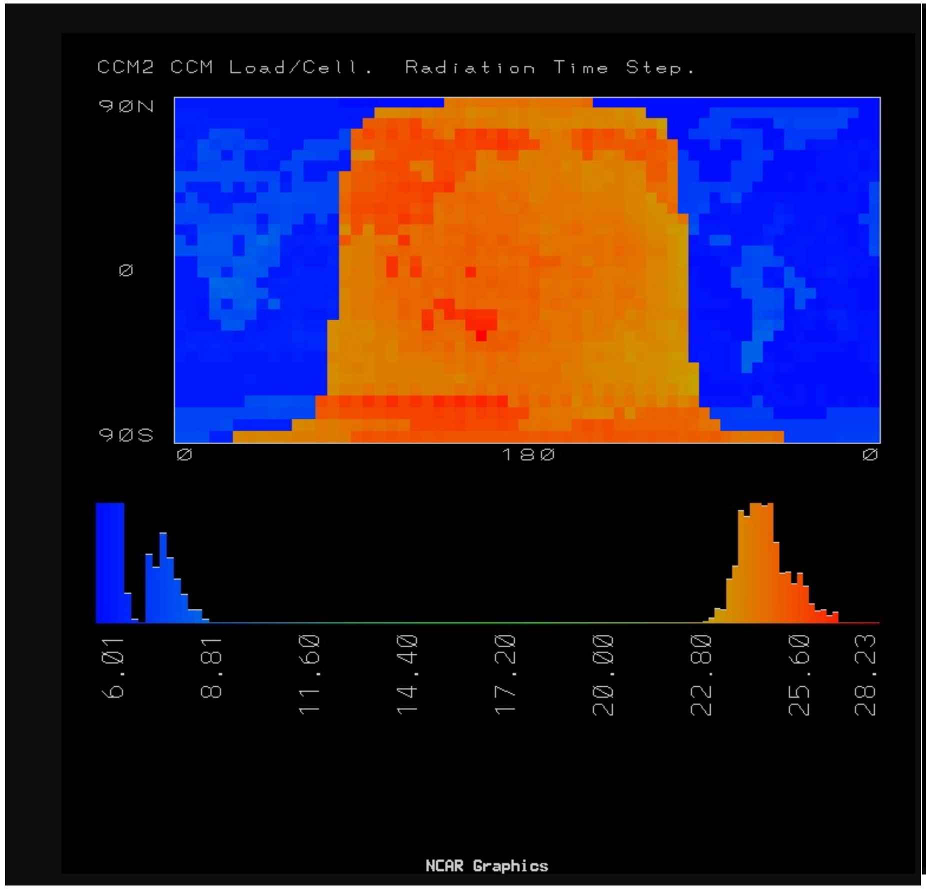
- In the absence of load imbalances, the simple mapping strategy illustrated in the following Figure can be used.
- It is clear from the figure that in this case, further agglomeration can be performed; in the limit, each processor can be assigned a single task responsible for many columns, thereby yielding an SPMD program.

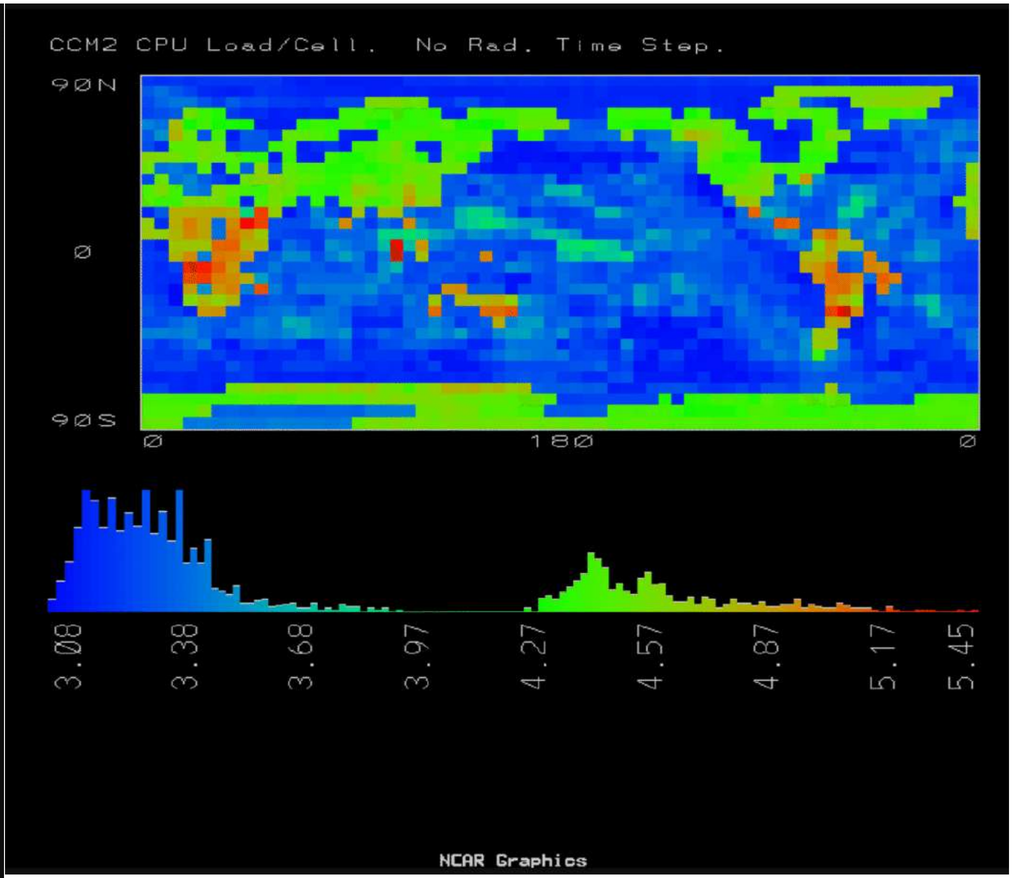


*Mapping in a grid problem in which each task performs the same amount of computation and communicates only with its four neighbors. The heavy dashed lines delineate processor boundaries. The grid and associated computation is partitioned to give each processor the same amount of computation and to minimize off-processor communication.*

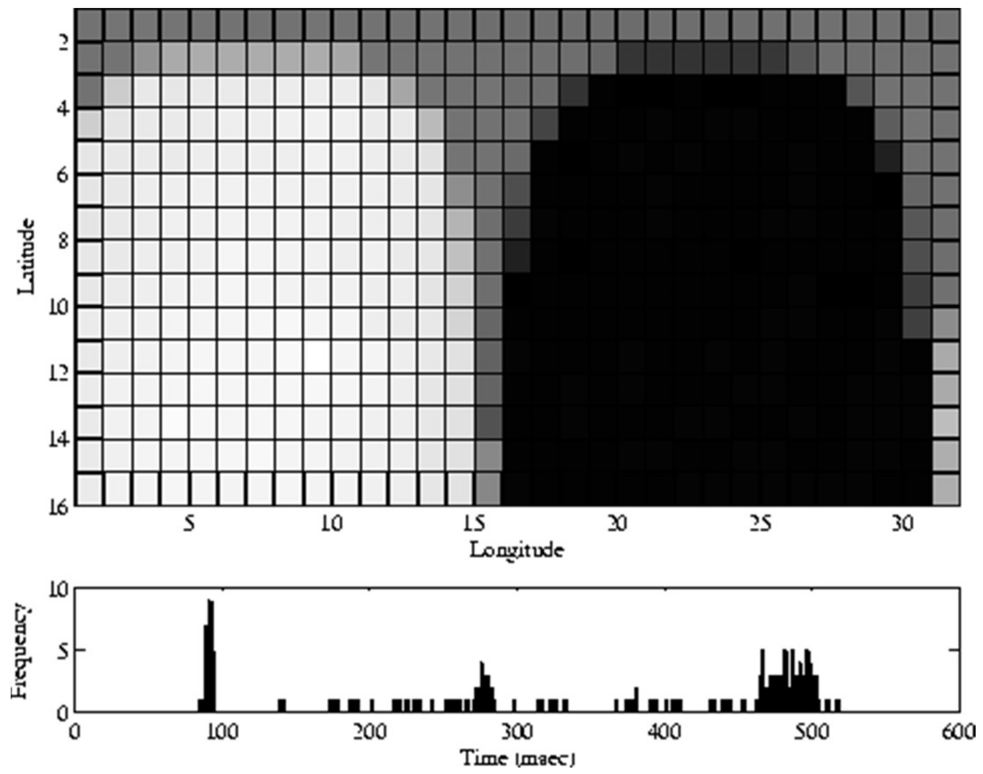
## Mapping

- This mapping strategy is efficient if each grid column task performs the same amount of computation at each time step. This assumption is valid for many finite difference problems but turns out to be invalid for some atmosphere models.
- The reason is that the cost of physics computations can vary significantly depending on model state variables. For example, **radiation calculations** are not performed at night, and **clouds** are formed only when humidity exceeds a certain threshold. **The sort of variation in computational load** that can result is illustrated in the following plate





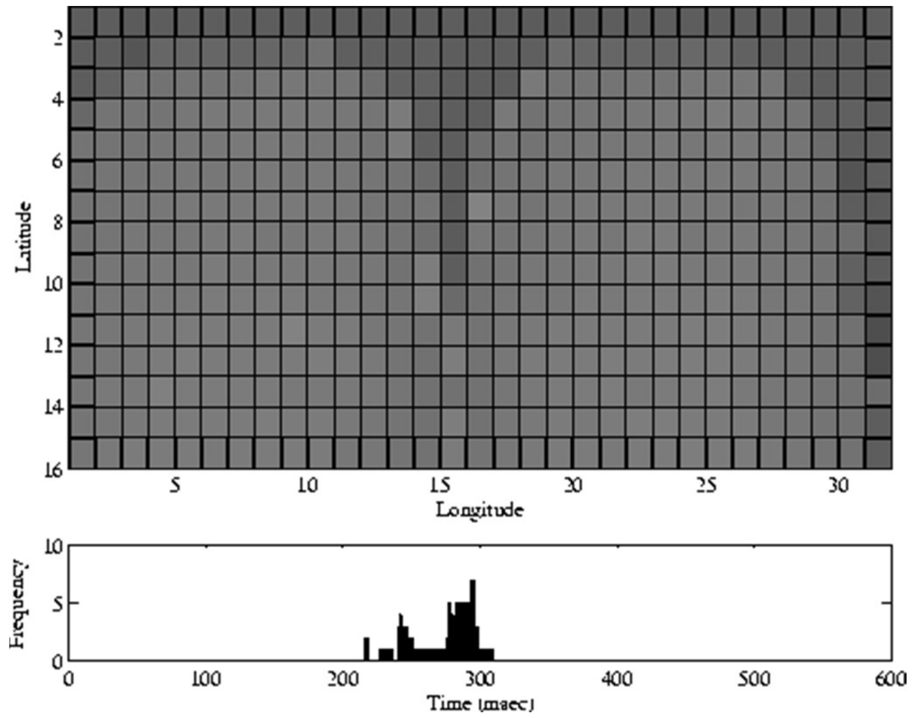
- Load distribution in an atmosphere model with a 64X128 grid. The figure shows per-point computational load at a single time step, with the histogram giving relative frequency of different load values. The left-hand image shows a time step in which **radiation time steps** are performed, and the right-hand image an ordinary time step. Diurnal, land/ocean, and local variations are visible. Images courtesy of J. Michalakes.



- ***Load distribution** in the physics component of an atmosphere model in the absence of load balancing.*

  *-In the top part of the figure, shading is used to indicate computational load in each of $16 \times 32$ processors. A strong spatial variation is evident. This effect is due to the night/day cycle (radiation calculations are performed only in sunlight); hence, there is a temporal variation also. The bottom part of the figure is a histogram showing the distribution of computation times, which vary by a factor of 5.*

  *-These results were obtained by using a parallel version of the National Center for Atmospheric Research (NCAR) Community Climate Model (CCM2) on the 512-processor Intel DELTA computer.*



- *Load distribution in the physics component of CCM2 when using a cyclic mapping. A comparison with the previous Figure shows that load imbalances are reduced significantly.*
- Empirical studies suggest that these load imbalances can reduce computational efficiency by 20 percent or more. In many circumstances, this performance loss may be regarded as acceptable. However, if a model is to be used extensively, it is worthwhile to spend time improving efficiency. One approach is to use a form of **cyclic mapping**: e. g., allocating each processor tasks from western and eastern and from northern and southern hemispheres.

(The reduction in load imbalance must be weighed against the resulting increase in communication costs.)

# Thank you!