IN5050:
Programming heterogeneous multi-core processors

Thinking Parallel

17/1 - 2022

Parallel algorithms

Have been developed for decades

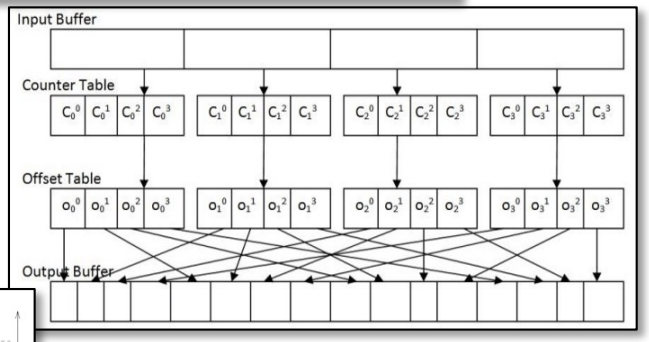
Algorithm examples ?

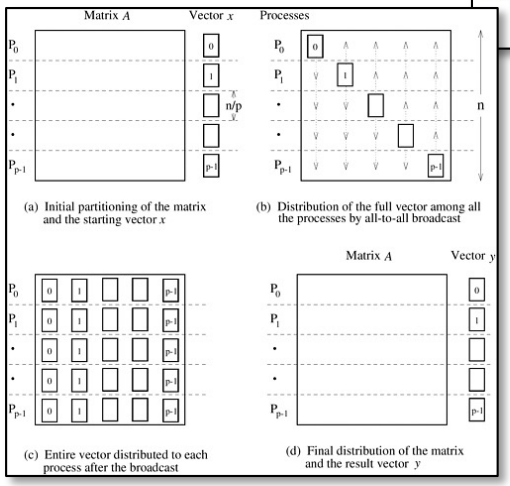
Parallel algorithms

Have been developed for decades

- Parallel search
- Parallel sort
- Prime numbers
- Dense matrix-vector multiplication
- Sparse matrix-vector multiplication
- Convex hull
- Parallel RANSAC (random sample consensus)
- …

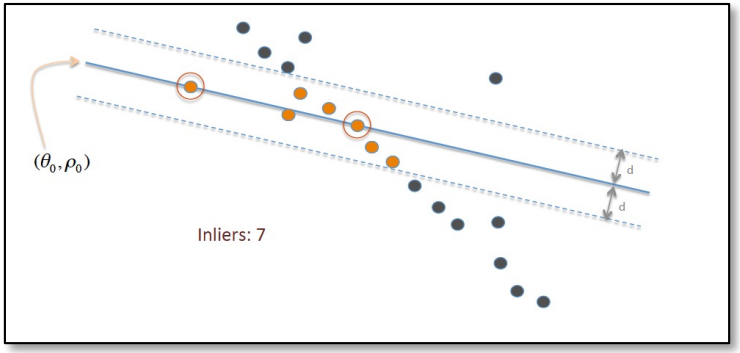
radix sort





rowwise 1D dMVM

line fitting



Parallel algorithms

Have been developed for decades

- Parallel search
- Parallel sort
- Prime numbers

$O(f(n) \Rightarrow O(f(n,p))$

- Dense matrix-vector multiplication

Data assignment

- Sparse matrix-vector multiplication
- Convex hull

memory latency

- Parallel RANSAC (random sample consensus)

randomness

- ...

Designing and Building Parallel Programs

- Ian Foster's framework proposal: http://www.mcs.anl.gov/~itf/dbpp/
  - be able to design simple parallel algorithms in a methodical fashion
  - be able to partition computations, using both domain and functional decomposition techniques
  - be able to use agglomeration as a means of reducing communication and implementation costs
  - be familiar with a range of load-balancing strategies
  - develop intuition as to what constitutes a good parallel algorithm
  - recognize design flaws that compromise efficiency or scalability
  - know how to recognize and implement both local and global, static and dynamic, structured and unstructured, and synchronous and asynchronous communication structures

Designing and Building Parallel Programs

Ian Foster's framework proposal

Be creative ...

..... but take a methodical approach

Methodical Design: **PCAM**

Partitioning

Communication

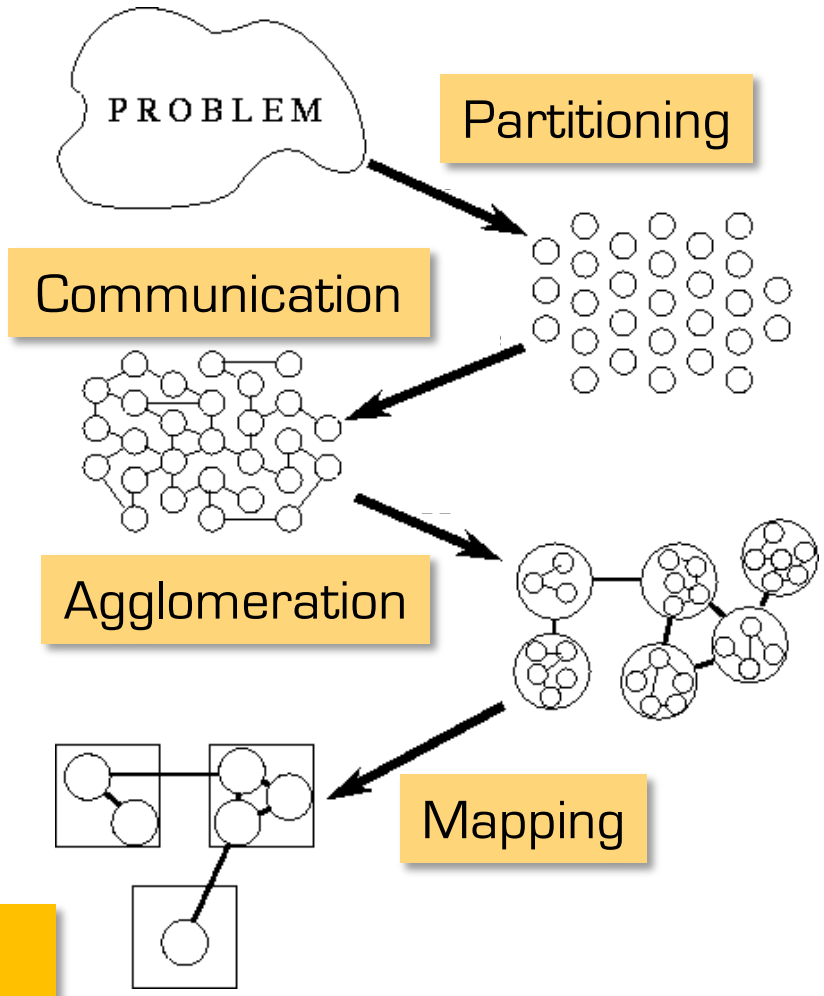
Agglomeration

Mapping

Methodical Design

- Most programming problems have several parallel solutions
- The best solution may differ from the best existing sequential one

- Outcome
  - Create, migrate and destroy tasks dynamically
  - use load-balancing techniques to control mapping
- Or
  - a SIMD program with one task per processor
  - however, agglomeration and mapping are one phase for SIMD



PCAM looks sequential

- it isn't
- explore alternatives right away
- avoid backtracking and complete redesign

Partitioning

Expose algorithmic opportunities for parallel execution

Which algorithmic opportunities?

- Provide the greatest flexibility
  - Later stages may invalidate opportunities
  - Typical reasons for invalidation:
    - Communication
    - Target architecture
    - Software engineering

Partitioning

Expose algorithmic opportunities for parallel execution

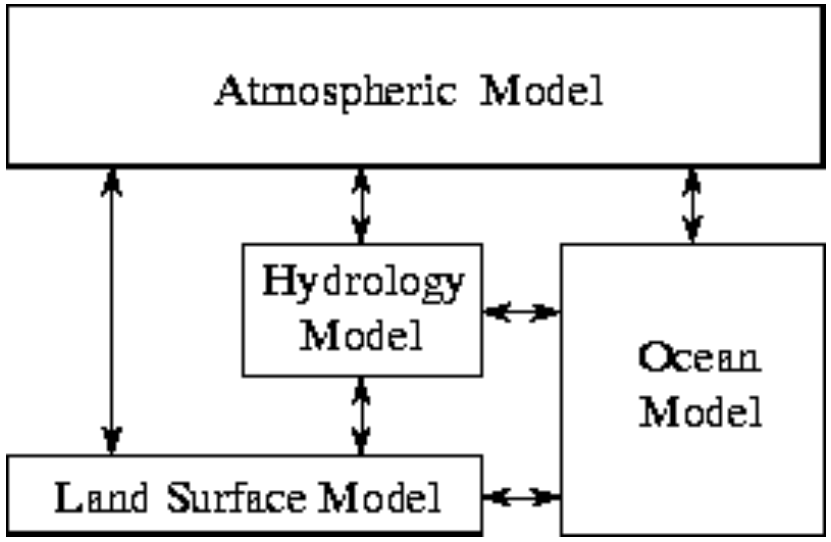
1. define a large number of small tasks
2. achieve a fine-grained decomposition

- Provide the greatest flexibility
  - Later stages may invalidate opportunities
  - Typical reasons for invalidation:
    - Communication
    - Target architecture
    - Software engineering

Revisit partitioning in agglomeration

Partitioning

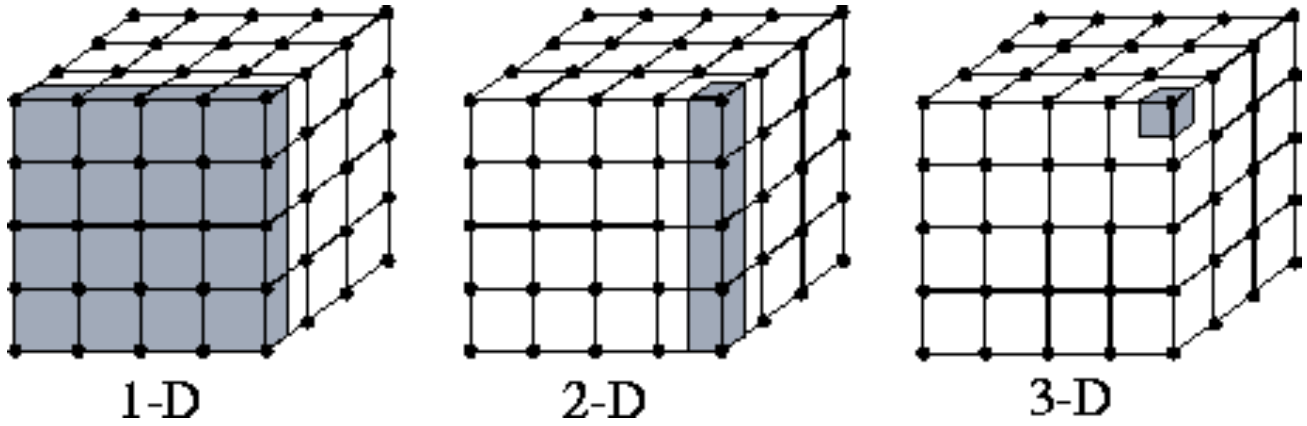
- Partition strategies
  - Functional decomposition
    - first decompose the computation
    - then deal with the data



- Modularity
- Pipelining (incl. parallel pipelines)
- Processing graphs
- Recursion

Partitioning

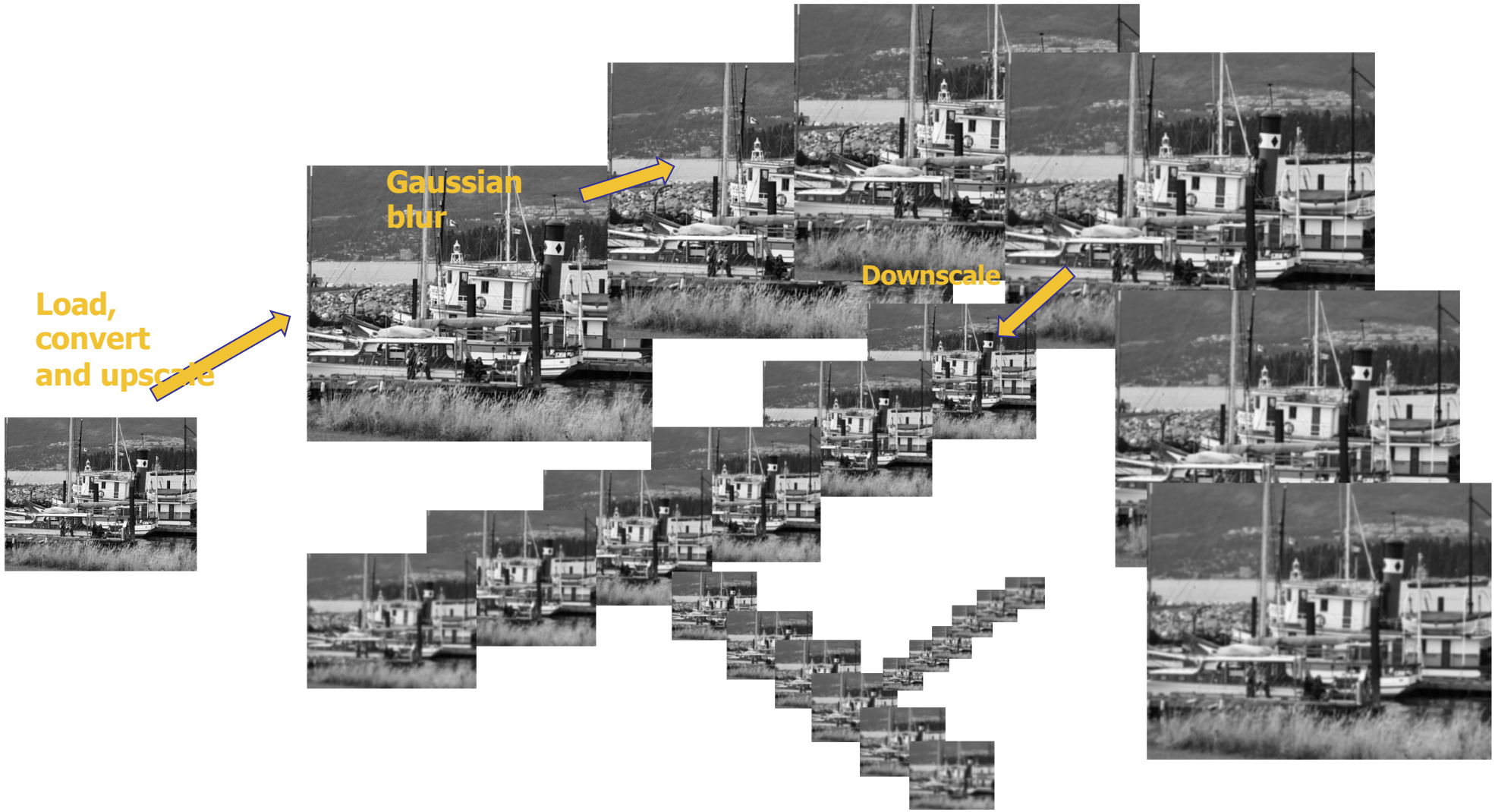
- Partition strategies
  - Domain (or data) decomposition
    - divides computation and data into small pieces
    - focus first on the data
    - partition it
    - finally, associate computation with data



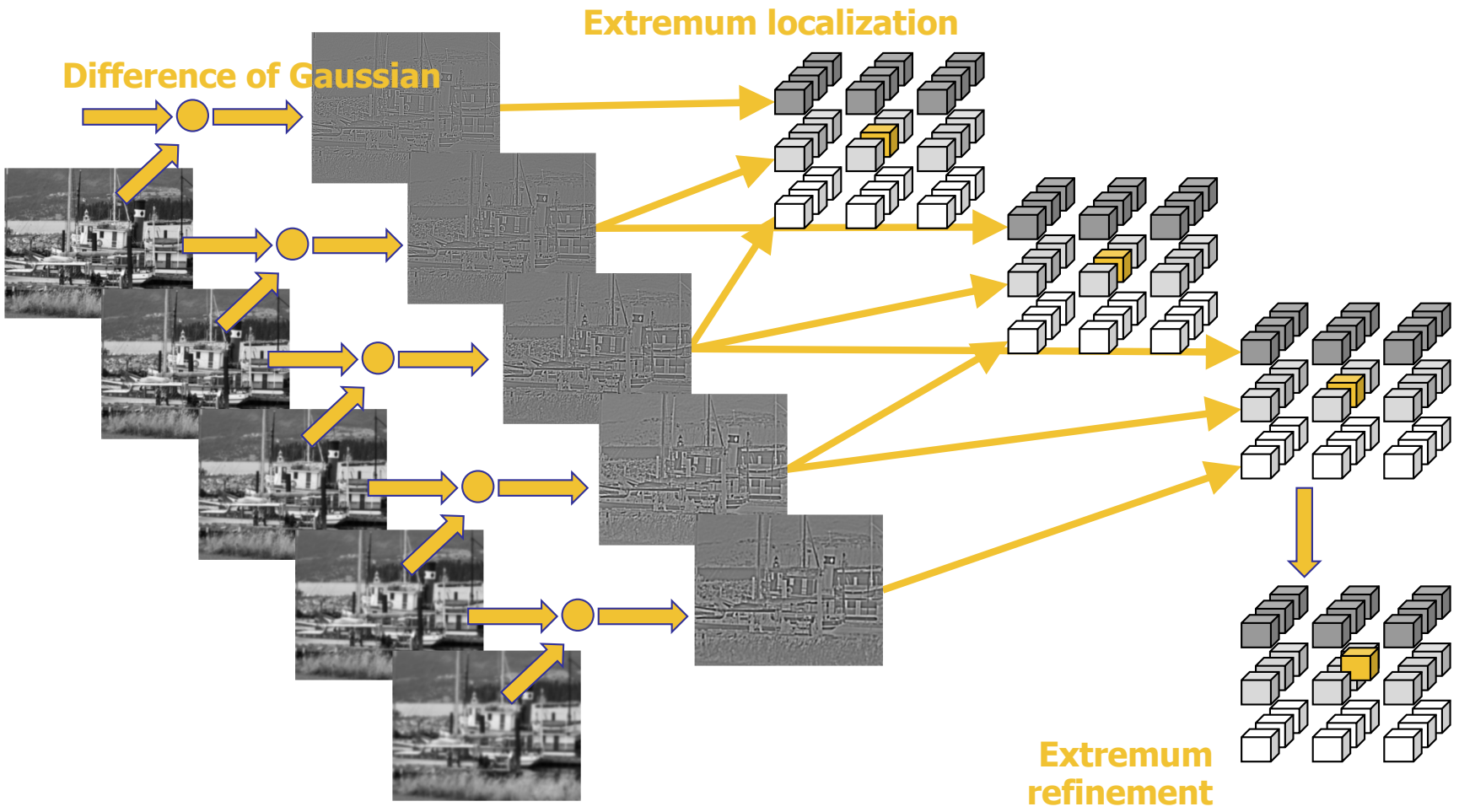
Partitioning

- Partition strategies
  - Domain decomposition
    - divides computation and data into small pieces
    - focus first on the data
    - partition it
    - finally, associate computation with data
  - Functional decomposition
    - first decompose the computation
    - then deal with the data
  - Complementary techniques
    - you can try both
    - even for parts of the same problem

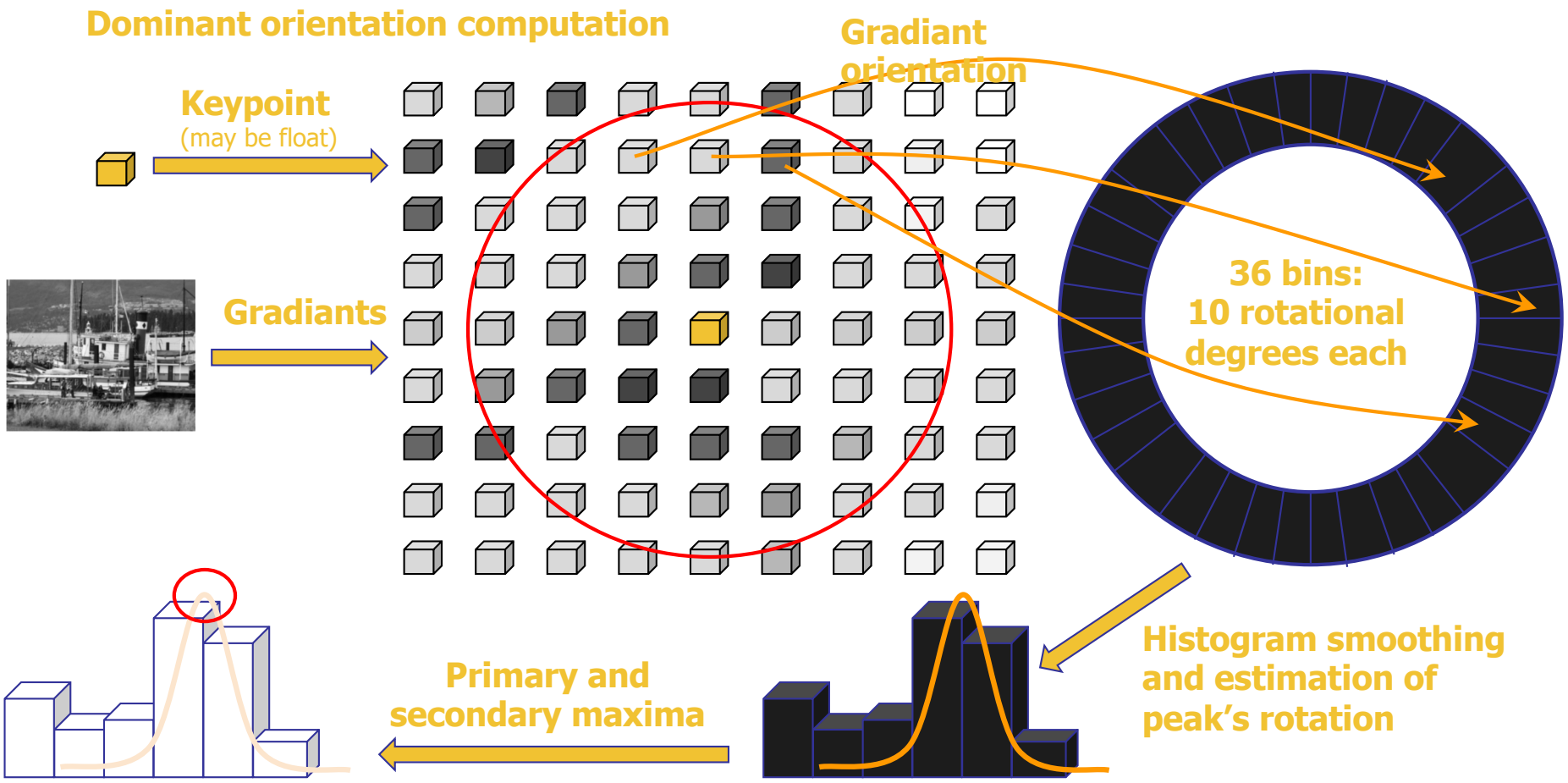
Combination example: SIFT (1/3)



Combination example : SIFT (2/3)



Combination example : SIFT (3/3)



Partitioning

Define disjoint sets of computation and data

- don't replicate computation
- don't replicate data
- revisit in communication

Partitioning Design Checklist

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
   - If this is not the case, 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
   - Remember to investigate both domain and functional decompositions

We may have to conclude that we haven't managed to find useful partition

Communication

Define the communication between partitions

Kinds of communication?

Communication

Define the communication between partitions

Tasks can rarely execute independently

Flow between tasks is specified in the communication phase

- Define a channel structure
  - consisting of producers and consumers
  - not necessarily implemented in software – could be memory reads and writes
  - avoid introducing unnecessary channels and communication
  - optimize performance by distributing communication operations over many tasks
  - organize communication operations in a way that permits concurrent execution

Communication

- Communication requirements for functional decomposition are often easy
  - correspond to the data flow between tasks

- In domain decomposition problems, communication requirements can be difficult
  - first partitioning data structures into disjoint subsets
  - associating operations with each datum
  - some operations that require data from several tasks usually remain
  - communication is then required
  - organizing this communication in an efficient manner can be challenging
  - even simple decompositions can have complex communication structures

Communication

- Local Communication
- Global Communication
- Unstructured and Dynamic Communication
- Asynchronous Communication

Communication Design Checklist

1. do all tasks perform about the same number of communication operations?
   - unbalanced communication requirements suggest a non-scalable construct
   - 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 non-scalable
   - try to use divide-and-conquer

4. is the computation associated with different tasks able to proceed concurrently?
   - if not, your algorithm is likely to be inefficient and non-scalable
   - consider whether you can reorder communication and computation operations

We may have to back to the specification

Agglomeration

Consider the usefulness of combining (agglomerating) tasks

Partitioning and communication provide abstract output

Their outputs are not specialized for any computers

Problems with result of C stage?

Agglomeration

Consider the usefulness of combining (agglomerating) tasks

Partitioning and communication provide abstract output

Their outputs are not specialized for any computers

- may be highly inefficient
  - e.g. creates many more tasks than there are processors
  - too small tasks
  - too many tasks
  - too much communication
  - too much scheduling overhead
  - …

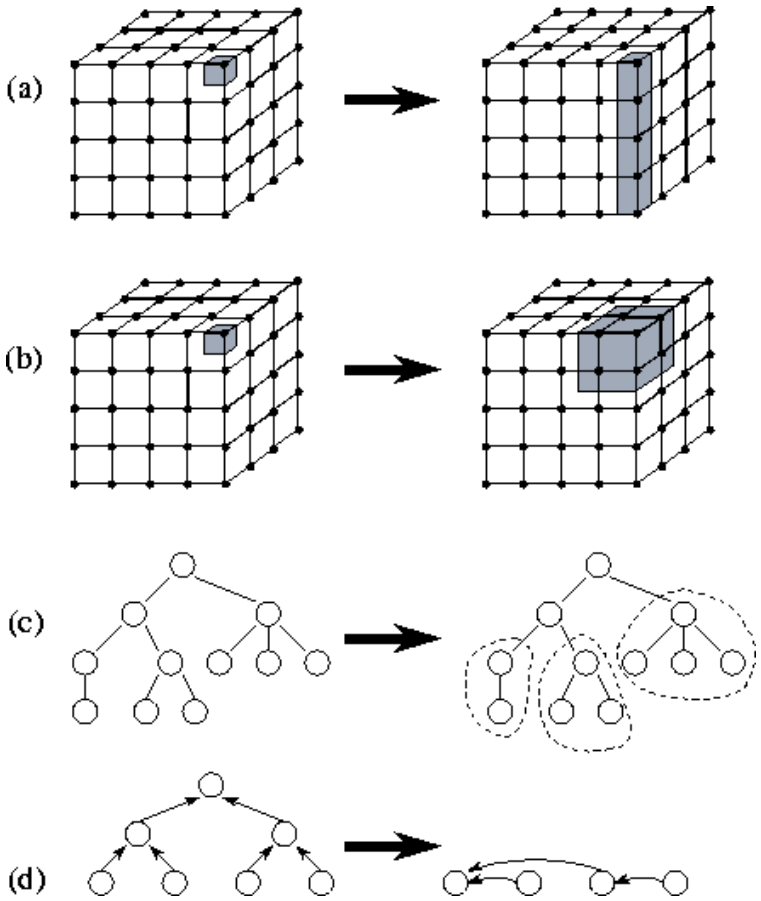
Agglomeration

Consider the usefulness of combining (agglomerating) tasks

Agglomeration moves from abstract to concrete

- revisit partitioning and communication
- consider whether it is useful to **combine** (agglomerate) tasks ...
- ... and **change mode** of communication
- determine whether it is worthwhile to **replicate** data and/or computation

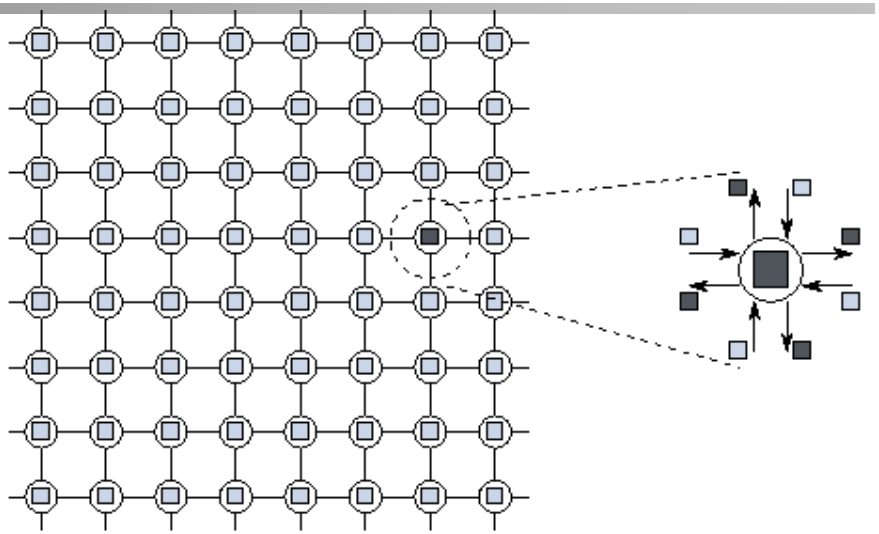
- number of tasks should still be greater than number of processors



Agglomeration

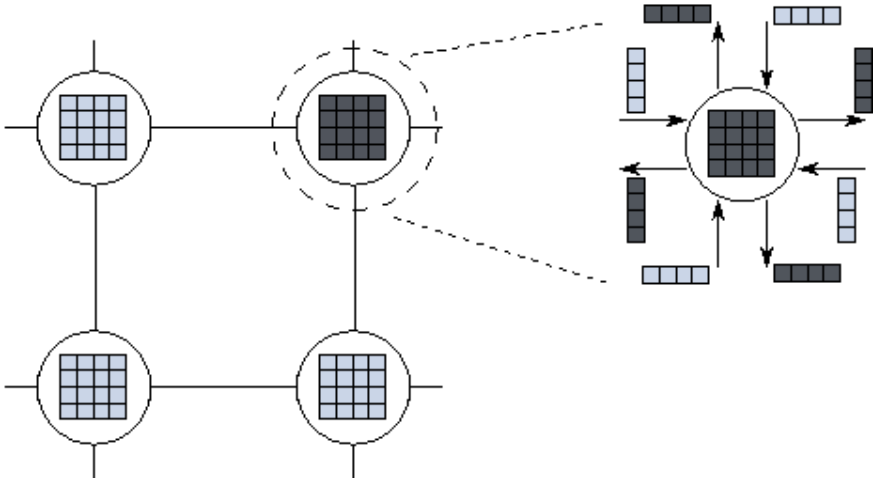
- Increasing Granularity
  - surface to volume effects
  - replicating computation
  - avoiding communication

(a)



- Preserving Flexibility
  - agglomerating can lead to design decisions & limit an algorithm's scalability unnecessarily

(b)



Agglomeration Design 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. 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?
   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
5. 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
6. 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
7. 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
8. 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

Mapping

Specify where each task executes

Specify where data is located

Architecture's influence on mapping?

Mapping

Mapping

Specify where each task executes

Specify where data is located

- Specify where each task executes
  - this does not apply:
    - on uniprocessors
    - on shared-memory computers with automatic task scheduling
- Sensible general-purpose mapping mechanisms (independent of the algorithms) for scalable parallel computers do not exist
- Mapping remains a difficult
  - mapping problem is known to be NP-complete

Mapping

Mapping

Specify where each task executes

Specify where data is located

- Mapping requires a goal
- Foster's goal
  - Minimize total execution time
- Other goals
  - Achieve a good trade-off between energy/price and execution time
  - Achieve real-time speeds
  - Maximize battery lifetime
  - ...

Mapping

Mapping

Specify where each task executes

Specify where data is located

- Mapping strategies
  - **increase concurrency**: place tasks that can execute concurrently on different processors
  - **increase locality**: place tasks that communicate frequently on the same processor

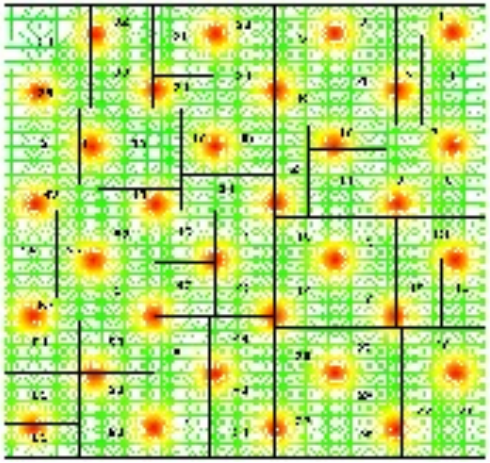
These strategies conflict

Resource limitations may restrict tasks per processor

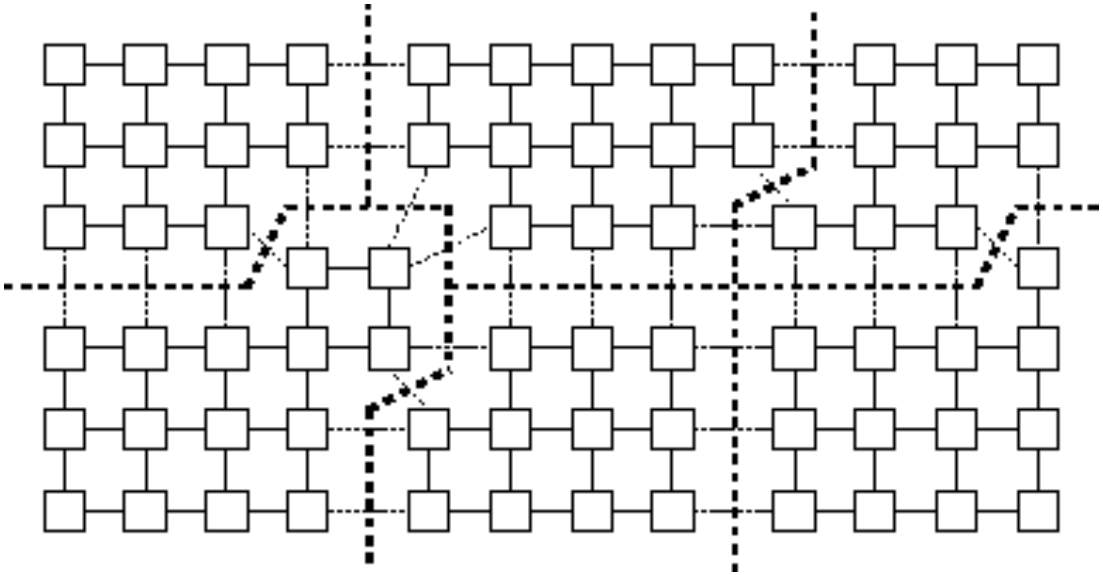
Design trade-offs exists

Mapping options

- Load Balancing
  - recursive bisection methods
  - probabilistic methods
  - cyclic mappings
  - local algorithms



- Task-Scheduling Algorithms
  - manager/worker
  - hierarchical manager/worker
  - decentralized schemes (work stealing)



Mapping Design Checklist

1. If considering a SIMD/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
   Performance can be problematic
2. If considering a design based on dynamic task creation and deletion, have you also considered a SIMD/SPMD algorithm?
   A SIMD/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 the tasks themselves, to the manager.
4. 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.
5. 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.