Designing Parallel Algorithms

Ned Nedialkov

Department of Computing and Software
McMaster University, Hamilton, Ontario
Canada

Outline

- Methodical design
- Partitioning
- Communication
- Conglomeration
- Mapping

Based on I. Foster, Designing and Building Parallel Programs [Chapter 2]
`http://www-unix.mcs.anl.gov/dbpp/`

Methodical design

Partitioning

Computation and data are decomposed into small tasks

Focus is on recognizing opportunities for parallel execution

Communication

Communication requirements, structures, and algorithms are determined

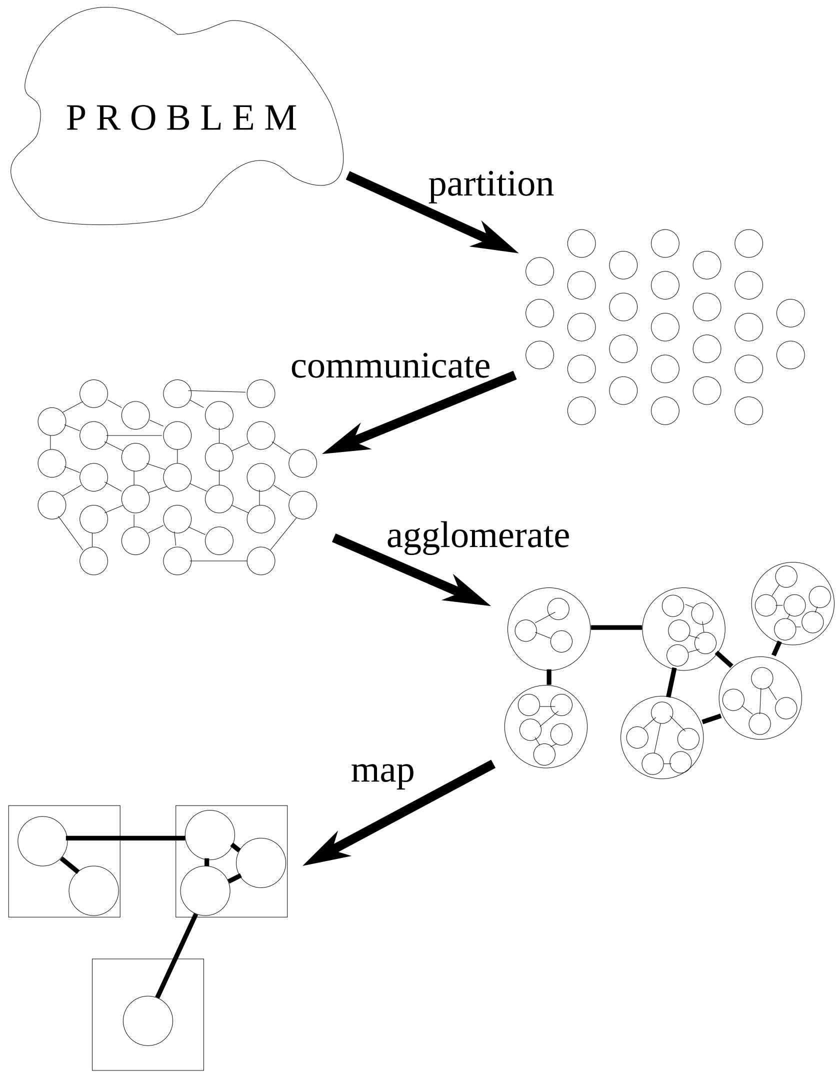
Agglomeration

Tasks are combined into larger tasks to improve performance

Mapping

Each task is assigned to a processor

Goal is to maximize processor utilization and minimize communication costs


PROBLEM
partition
communicate
agglomerate
map

Partitioning

Focus is on recognizing opportunities for parallel execution

Computation and the data operated on by this computation are decomposed

Define a number of small tasks in order to yield what is termed a fine-grained decomposition of a problem

A good partition divides into pieces both computation and data

We try to avoid replicating computation and data

That is, we seek to define tasks that partition both computation and data into disjoint sets

Domain decomposition

- partition the data
- determine communication associated with the partition

Functional decomposition

- decompose the computation into tasks
- decompose the data based on these tasks

Common in problems where there no obvious data structures to partition

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 may have little flexibility in subsequent design stages

✓ Does your partition avoid redundant computation and storage requirements?

  If not, it may not be scalable

✓ Are tasks of comparable size?

✓ Does the number of tasks scale with problem size?

✓ Have you identified several alternative partitions?

Communication

Tasks generated by a partition are intended to execute concurrently

The computation to be performed in one task will typically require data associated with another task

Data must then be transferred between tasks, to allow computation to proceed

This information flow is specified in the communication phase of a design

Local communication: each task communicates with a small set of other tasks (its "neighbors")

Global communication: requires each task to communicate with many tasks

Structured communication: a task and its neighbors form a regular structure, e.g. a tree or grid

Unstructured communication: communication pattern may be complex and irregular

Static communication: the communication pattern does not change over time

Dynamic communication: the communication pattern is determined at runtime and may be variable

Example: Jacobi vs. Gauss-Seidel

Consider Jacobi's method

A multidimensional grid is repeatedly updated by replacing the value at each point with some function of the values at a small, fixed number of neighboring points

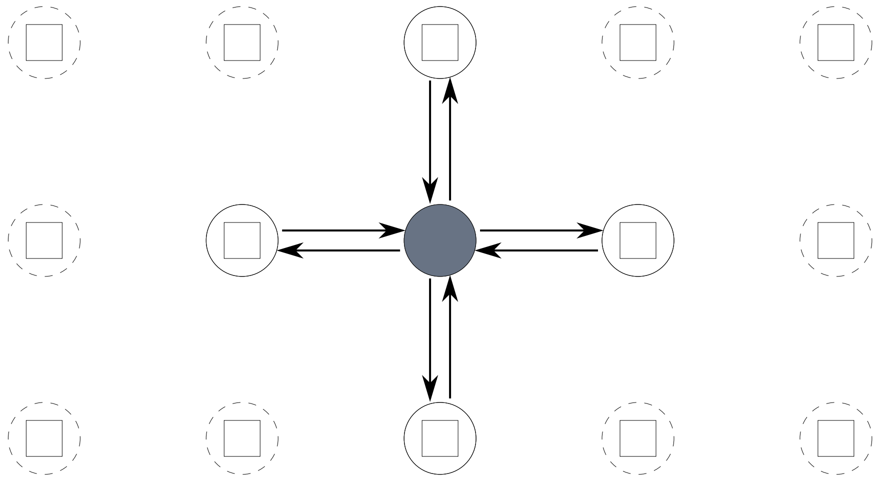
The set of values required to update a single grid point is called that grid point's stencil

For example, the following expression uses a five-point stencil to update each element of a two-dimensional grid:

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

k is iteration number

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



For each boundary point, we have

```
for $k = 0, 1, \ldots$
    send $X_{i,j}^{(k)}$ to each neighbor
    receive $X_{i-1,j}^{(k)}$, $X_{i+1,j}^{(k)}$, $X_{i,j-1}^{(k)}$, $X_{i,j+1}^{(k)}$
    compute $X_{i,j}^{(k+1)}$
```

In serial, Gauss-Seidel (G-S) is often more efficient than Jacobi

G-S is more difficult to parallelize

Consider the simple G-S scheme

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

In average, $\approx N/2$ points of an $N \times N$ grid can be updated in parallel

Red-black ordering

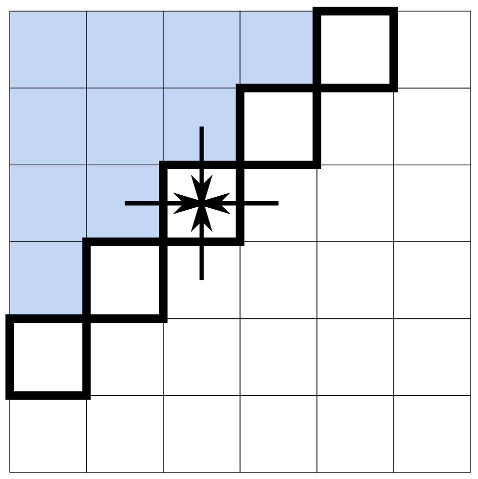
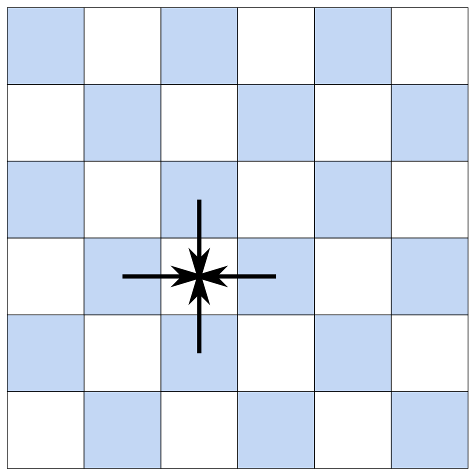
One can update first the odd-numbered elements and then the even-numbered elements of an array

Each update uses the most recent information

Updates to the odd-numbered [resp. even-numbered] points are independent and can proceed concurrently

This is red-black ordering: points can be thought of as being colored as on a chess board, red (odd) or black (even)

Points of the same color can be updated concurrently

Two finite difference update strategies

Shaded grid points have already been updated to step $k + 1$

Unshaded grid points are still at step k

Left: simple G-S; the update proceeds in a wavefront from the top left corner to the bottom right

Right: red-black update scheme; all the grid points at step k can be updated concurrently

The Jacobi update strategy is efficient in parallel, but inferior numerically

The red-black scheme combines the advantages of G-S and Jacobi

Global communication

Global communication: many tasks must participate

One should try to avoid creating too many communications or restricting opportunities for concurrent execution

Example

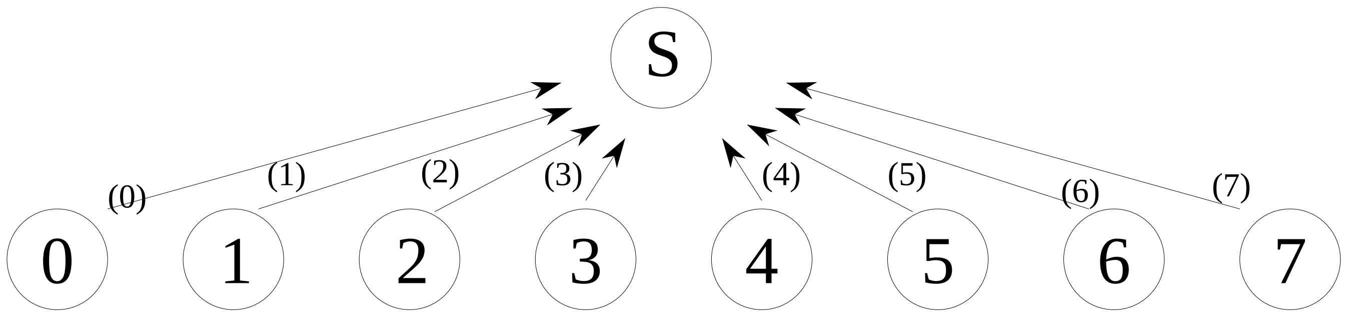
Consider a parallel reduction operation that sums N values distributed over N tasks

$$S = \sum_{i=0}^{N-1} X_i$$

Assume a single "manager" task requires the result S of this operation

Taking a purely local view of communication, we recognize that the manager requires values X_0, X_1, etc., from tasks 0, 1, etc.

Hence, we could define a communication structure that allows each task to communicate its value to the manager independently



The manager can receive and sum only one number at a time

This approach takes $O(N)$ time to sum N numbers—not a very good parallel algorithm!

- algorithm is centralized: it does not distribute computation and communication
- algorithm is sequential: it does not allow multiple computation and communication operations to proceed concurrently

We must address both these problems to develop a good parallel algorithm

Distributing communication and computation

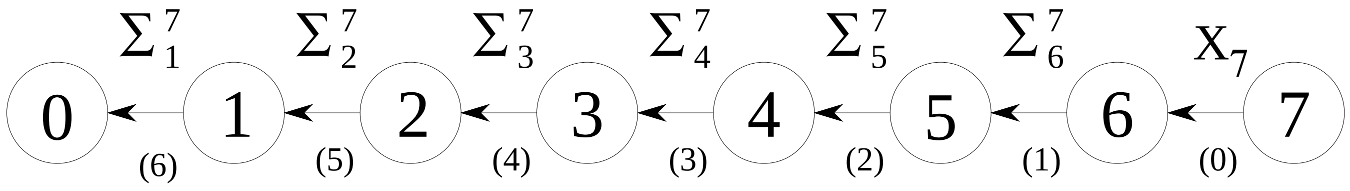
Consider distributing the computation and communication associated with the summation

We can distribute the summation of the N numbers by making each task i, $0 < i < N-1$, compute the sum:

$$S_i = X_i + S_{i-1}$$

The N tasks are connected in a one-dimensional array

Task $N-1$ sends its value to its neighbor in this array



Tasks 1 through $N-2$ 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

This approach distributes the $N-1$ communications and additions, but permits concurrent execution only if multiple summation operations are to be performed

The array of tasks can then be used as a pipeline, through which flow partial sums

A single summation still takes $N-1$ steps

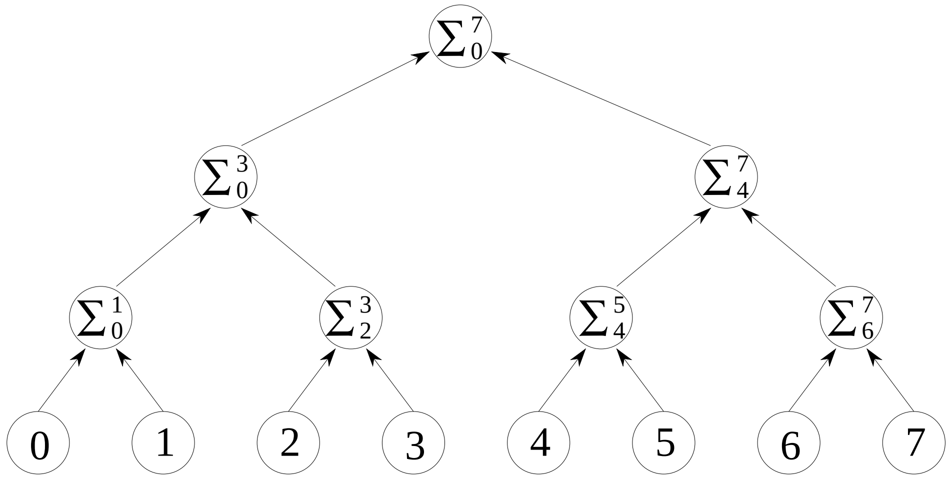
Uncovering concurrency: divide and conquer

Divide and conquer: divide into two or more simpler problems of roughly equivalent size (e.g., summing $N/2$ numbers)

Apply recursively

Effective in parallel computing when the subproblems generated by problem partitioning can be solved concurrently

We can do the summation as



Summations at the same level can be done in parallel

$O(\log N)$ communication steps

We distributed $N - 1$ communication and computation operations required to perform the summation

We modified the order in which these operations are performed so that they can proceed concurrently

The result is a regular communication structure in which each task communicates with a small set of neighbors

Unstructured and dynamic communication

The previous examples are all of static, structured communication

In practice communication pattern

- may be considerably more complex and
- may change over time

Communication design checklist

- ✓ Do all tasks perform about the same number of communication operations?

  Unbalanced communication requirements suggest a non-scalable construct

  See if communication operations can be distributed more equitably

- ✓ Does each task communicate only with a small number of neighbors?

  If each task must communicate with many other tasks, consider formulating this global communication in terms of local communication

- ✓ Are communication operations able to proceed concurrently?

- ✓ 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

  You may also wish to revisit your problem specification (as was done in moving from a simple G-S to a red-black algorithm)

Agglomeration

We move from the abstract toward the concrete

From the first two stages, we have an abstract algorithm; not specialized for efficient execution on any particular parallel computer

It may be highly inefficient if, for example, it creates many more tasks than there are processors on the target computer, and this computer is not designed for efficient execution of small tasks

We revisit decisions made in the partitioning and communication phases to obtain an algorithm that will execute efficiently on some class of parallel computer

We consider whether it is useful to combine, or agglomerate, tasks identified by the partitioning phase, to provide a smaller number of tasks, each of greater size

We also determine whether it is worthwhile to replicate data and/or computation

Increasing granularity

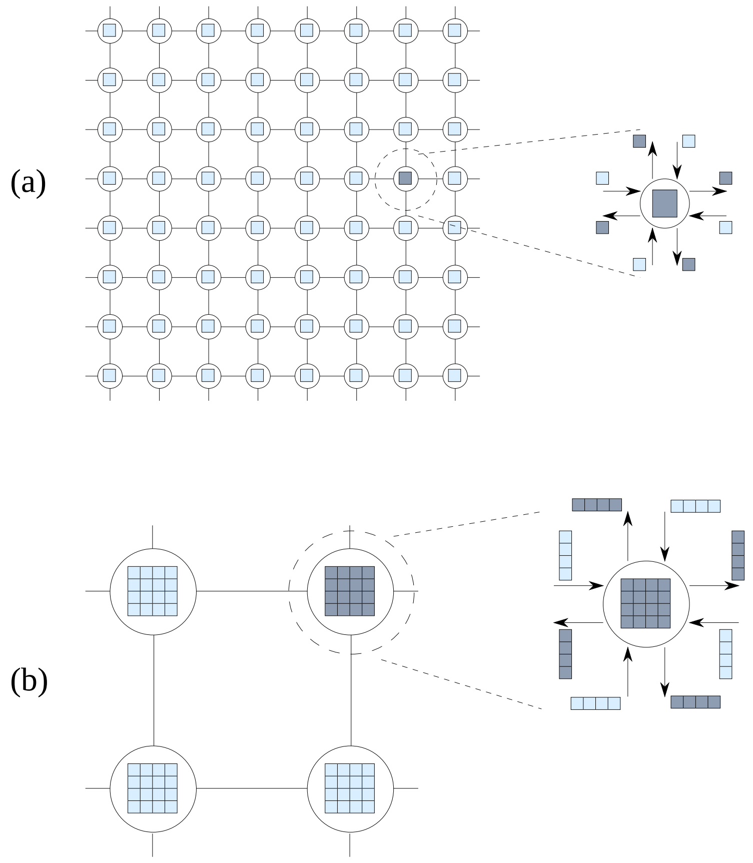
In the partitioning phase, our efforts are focused on defining as many tasks as possible

Forces us to consider a wide range of opportunities for parallel execution

Large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm

Critical issue is communication costs

Each communication incurs not only a cost proportional to the amount of data transferred, but also a fixed startup cost



Effect of increased granularity on communication costs in a two-dimensional finite difference problem, five-point stencil
(a) 64 tasks, each responsible for a single point; 4 communications per task, $64 \times 4 = 256$ communications, 256 data points transferred
(b) 4 tasks, each responsible for 16 points; $4 \times 4 = 16$ communications, $16 \times 4 = 64$ data points transferred

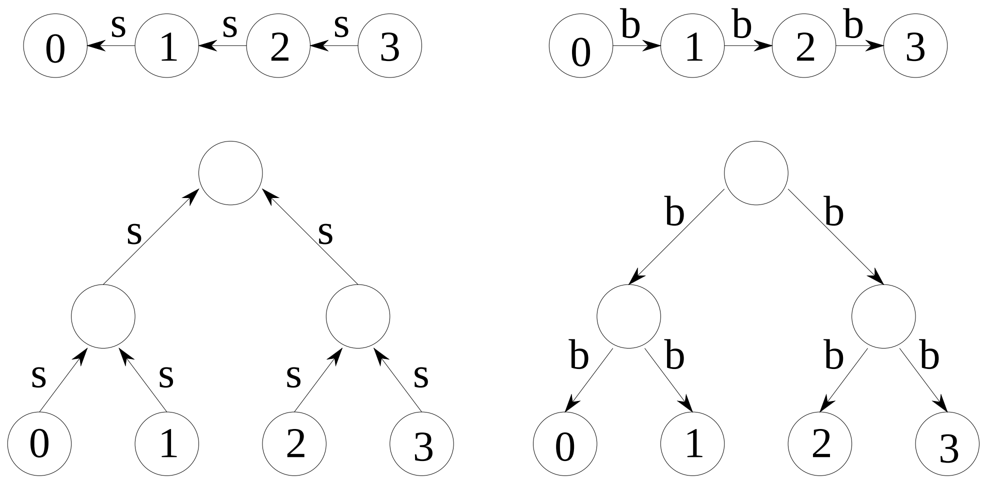
Replicating computation

We can sometimes trade off replicated computation for reduced communication requirements and/or execution time

Consider a variant of the summation problem in which the sum must be replicated in each of the N tasks that contribute to the sum

We can use a ring- or tree-based algorithm to compute the sum in a single task, and then broadcast the sum to each of the N tasks

The broadcast can be performed using the same communication structure as the summation



The complete operation can be performed in either $2(N-1)$ or $2N\log N$ steps

These algorithms are optimal in the sense that they do not perform any unnecessary computation or communication

However, there are alternative algorithms that execute in less elapsed time, although at the expense of unnecessary computation and communication

Basic idea: perform multiple summations concurrently, with each concurrent summation producing a value in a different task

Approach 1. All tasks are connected in a ring, 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 $O((N-1)^2)$ redundant additions and $O((N-1)^2)$ unnecessary communications

However, the summation and broadcast complete in $N-1$ rather than $2(N-1)$ steps

Hence, this strategy is faster, if the processors would otherwise be idle waiting for the result of the summation

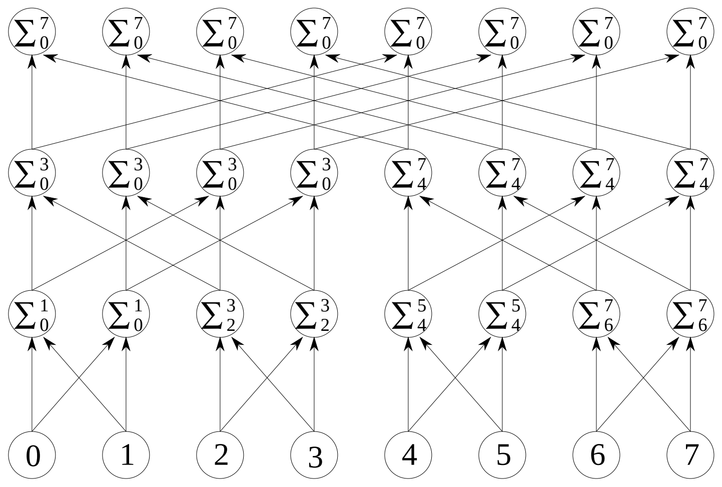
Approach 2. The tree summation algorithm can be modified in a similar way to avoid the need for a separate broadcast

That is, multiple tree summations are performed concurrently so that after $\log N$ steps each task has a copy of the sum

One might expect this approach to result in $O(N^2)$ additions and communications, as in the ring algorithm

However, in this case we can exploit redundancies in both computation and communication to perform the summation in just $O(N \log N)$ operations

The resulting communication structure is termed a butterfly


Σ_0^7 Σ_0^7 Σ_0^7 Σ_0^7 Σ_0^7 Σ_0^7 Σ_0^7 Σ_0^7
Σ_0^3 Σ_0^3 Σ_0^3 Σ_0^3 Σ_4^7 Σ_4^7 Σ_4^7 Σ_4^7
Σ_0^1 Σ_0^1 Σ_2^3 Σ_2^3 Σ_4^5 Σ_4^5 Σ_6^7 Σ_6^7
0 1 2 3 4 5 6 7

Agglomeration design checklist

- ✓ Has agglomeration reduced communication costs by increasing locality?
- ✓ If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs
- ✓ 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?
- ✓ Does the number of tasks still scale with problem size?
- ✓ 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.

Mapping

We specify where each task is to execute

Two strategies:

1. Place tasks that are able to execute concurrently on different processors, so as to enhance concurrency
2. Place tasks that communicate frequently on the same processor, so as to increase locality.

Resource limitations may restrict the number of tasks that can be placed on a single processor

The mapping problem is known to be NP complete

Goal is to minimize total execution time

Mapping decisions seek to balance conflicting requirements for equitable load distribution and low communication costs