PARALLEL PROGRAM DESIGN

Course "Parallel Computing"

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

`Wolfgang.Schreiner@risc.jku.at`

`http://www.risc.jku.at`

Designing Parallel Programs

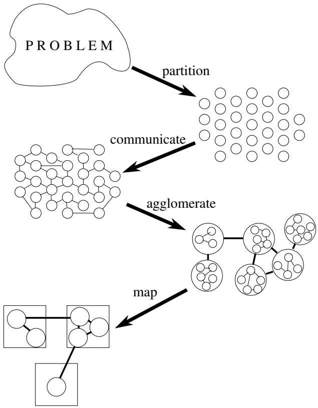
Ian Foster: “Designing and Building Parallel Programs”.

- First consider machine-independent (algorithmic) issues.
  - Concurrency.
  - Scalability.
- Later deal with machine-specific (performance) aspects.
  - Locality.
  - Placement.

A methodological approach in multiple stages.

The PCAM Approach

- Partitioning.
  - Decompose computation and data.
  - Exhibit opportunities for parallelism by creating many small tasks.
- Communication.
  - Analyze data dependencies.
  - Determine structure of commmunication and coordination.
- Agglomeration.
  - Combine tasks to bigger tasks.
  - Improve performance of execution on real computers.
- Mapping.
  - Assign tasks to processors.
  - Maximize utilization and minimize communication.



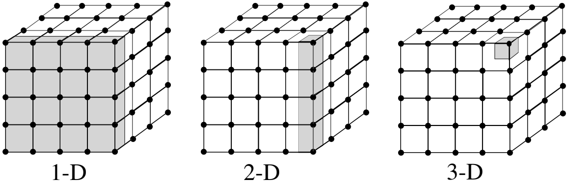
Partitioning

Expose opportunities for parallelism.

- Construct fine-grained decomposition of problem.
  - Domain/data decomposition:
    - Partition data, associate computation to data.
  - Functional/task decomposition:
    - Partition computation, associate data to computation.
- Complementary approaches.
  - Should be both considered.
  - Can lead to alternative algorithms.
  - Can be applied to different parts of problem.
- Avoid replication of computation or data.
  - May be introduced later to reduce communication overhead and to increase the granularity of tasks.

Domain Decomposition

Focus on the decomposition of the data.

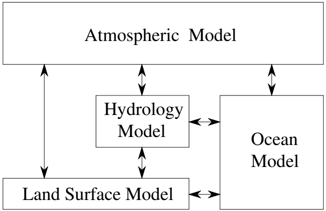


- Divide data into small pieces and associate computation.
  - If computation requires several, associate to “main” piece.
  - Communication is required for access to the other pieces.
- Resulting tasks should be of roughly the same size.
  - Otherwise load balancing may become difficult.
- Prefer finer decomposition over coarse ones.
  - Small tasks may be agglomerated in later stage.

Typical for problems with large central data structures.

Functional Decomposition

Focus on the decomposition of the computation.



- Decompose according to the algorithmic structure.
  - Independent computational blocks.
  - Independent loop iterations.
  - Independent (recursive) function invocations.
- Determine data requirements of each task.
  - If requirements overlap, communication is required.

Typical for problems without central data structures.

Partitioning Design Checklist

- Is number of tasks large enough?
  - Order of magnitude larger than processor number.
  - Keeps flexibility for further stages.
- Does number of tasks scale with problem size?
  - Larger problems can be solved with more processors.
- Are the tasks of comparable size?
  - Otherwise load balancing may become difficult.
- Are redundant computations and data avoided?
  - Otherwise scalability may suffer.
- Have alternative partitions been considered?
  - Try both domain and functional decomposition.

Do we have sufficient concurrency?

Communication

Specify flow of information between tasks.

- Describe communication structure by “channels”.
  - Connections between those tasks that produce data and those that consume them.
  - Typically easy to determine for functional decomposition from data flow between tasks.
  - May be complex to determine for domain decomposition due to data dependencies.
- Analyze the usage of channels.
  - Number and sizes of messages flowing through channels.
  - Temporal relationship/dependencies between messages flowing through different channels.

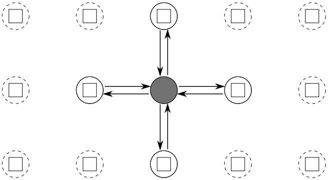
Also a healthy exercise for shared memory programs.

Types of Communication

- Local versus global:
  - Communication with a small set of tasks (“neighbors”) or with many other tasks.
- Structured versus unstructured:
  - Communication forms a regular structure (tree, grid, . . .) or an arbitrary graph.
- Static versus dynamic:
  - Identity of communication partners is known in advance and does not change or depends on runtime data and may vary.
- Synchronous versus asynchronous:
  - Producers and consumers cooperate in data transfer or consumer may acquire data without producer cooperation.

Local Communication

Example: Jacobi finite differences method.



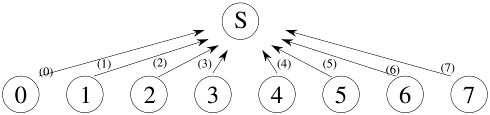
$$X_{i,j}^{t+1} = \frac{1}{8}\left(4X_{i,j}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t} + X_{i,j-1}^{t} + X_{i,j+1}^{t}\right)$$

```
for t=0 to T-1 do
  send X(i,j) to each neighbor
  receive X(i-1,j), X(i+1,j), X(i,j-1), X(i,j+1) from neighbors
  update X(i,j)
end
```

Global Communication

Example: parallel reduction operation.

$$S = \sum_{i=0}^{n} X_i$$

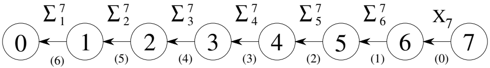


- Centralized algorithm:
  - Single task becomes bottleneck of communication and computation.
- Sequential algorithm:
  - Additions are performed one after each other.

Global Communication

Example: parallel reduction operation.

$$\sum_{i=j}^{n} X_i = X_j + \sum_{i=j+1}^{n} X_i$$

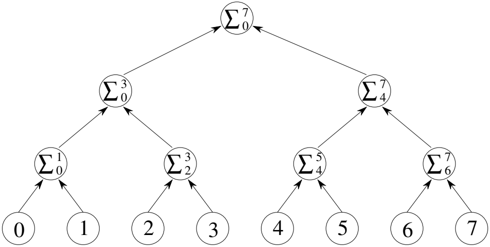


- Decentralized algorithm:
  - Communication/computation are distributed among tasks.
- But still a sequential algorithm.

Global Communication

Example: parallel reduction operation.

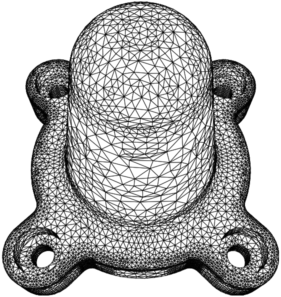
$$\sum_{i=j}^{j+k} X_i = \left(\sum_{i=j}^{j+\lfloor k/2 \rfloor} X_i\right) + \left(\sum_{i=j+\lfloor k/2 \rfloor+1}^{j+k} X_i\right)$$



- Decentralized and parallel algorithm:
  - Up to $k/2$ additions can be performed in parallel.

Unstructured/Dynamic Communication

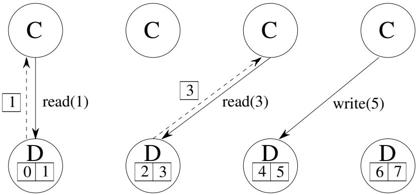
Example: finite element method.



- Mesh of points representing a physical object.
  - Simulation of, e.g., the impact of force on the object.
  - Shape of the mesh is modified by the impact.
- Domain decomposition.
  - Unstructured communication: mesh is irregular.
  - Dynamic communication: mesh changes.

Asynchronous Communication

Example: management of a shared data structure.



- A set of “data tasks” manages a shared data structure.
  - Data structure is distributed among tasks.
- A set of “computing tasks” produce and consume data.
  - Exchange of messages between computing tasks and data tasks for reading and writing the data structure.

Consumption of data decoupled from their production.

Communication Design Checklist

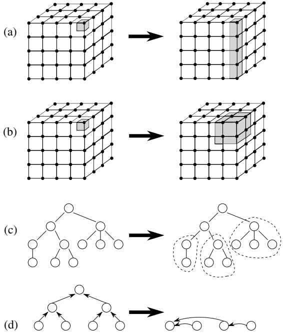
- Do all tasks perform the same amount of communication?
- Does each task communicate only with a few neighbors?
- Can the communication operations proceed concurrently?
- Can the computation operations proceed concurrently?

Do we have the potential for scalability?

Agglomeration

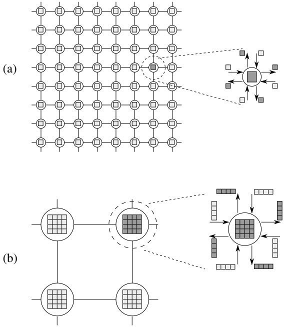
In the previous phases we have developed a parallel algorithm.

- Algorithm not efficiently executable.
  - Large number of small tasks.
  - Large amount of communication.
- Combine tasks to larger tasks.
  - Increase the granularity of tasks.
    - Granularity: the ratio of computation to communication.
  - Still retain design flexibility.
    - Sufficiently many tasks for scalability and mapping flexibility.
  - Reduce engineering costs.
    - Avoid effort of parallelization where it does not pay off.



Increasing Granularity: Surface to Volume

- Before: granularity $1/4 = 0.25$.
  - 1 local computation operation.
  - 4 data items sent.
- After: granularity $16/16 = 1$.
  - 16 local computation operations.
  - 16 data items sent.
- Surface to Volume Effect
  - Typical for domain decomposition.
  - Communication proportional to "surface" of subdomain.
  - Computation proportional to "volume" of subdomain.
  - Surface grows slower than volume.
    - Square: $S/V = 4a/a^2 = 4/a$.

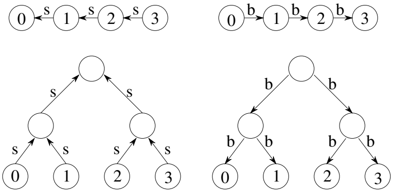


Decreasing surface-to-volume ratio increases granularity.

Increasing Granularity: Replicating Computation

Communication may be decreased by replicating computation.

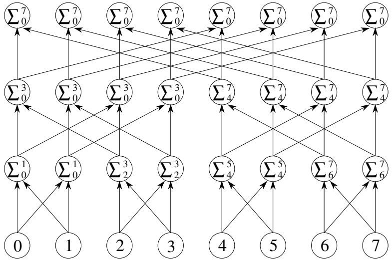
Example: two algorithms computing a global sum in N tasks.



Time $2(N-1)$ resp. $2\log_2 N$ for performing $N-1$ additions.

Increasing Granularity: Replicating Computation

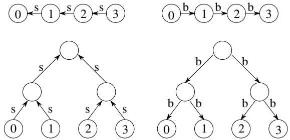
A replicating algorithm computing a global sum in N tasks.

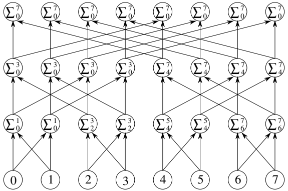


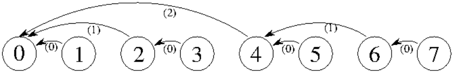
Time $\log_2 N$ for performing $N \log N$ additions.

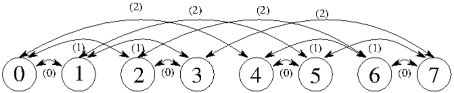
Increasing Granularity: Avoiding Communication

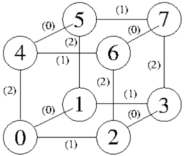
Agglomerate tasks that cannot execute concurrently.











Only N agglomerated tasks are needed.

Retaining Design Flexibility

Do not “over-agglomerate”.

- Goal is not a fixed number of tasks.
  - Task number should grow with problem and machine size.
  - Algorithm should remain scalable.
- Goal is not one task per processor.
  - There shold be still multiple tasks per processor.
  - If one task is blocked, another one may execute and keep the processor busy.

Agglomeration should not “hardwire” the algorithm to a fixed problem and machine size.

Reducing Engineering Costs

- Try to avoid extensive code changes.
  - One partitioning/agglomeration may be much more difficult to implement than another.
- Try to avoid extensive data structure changes.
  - Conversions from/to data structures given by the context of the parallel application may be cumbersome.

Consider also the costs of development in relation to the expected performance gains.

Agglomeration Design Checklist

- Has communication been reduced (granularity increased)?
- Does computation replication outweigh its costs?
- Does data replication not limit scalability?
- Have tasks still similiar sizes?
- Is there still sufficient concurrency?
- Does the number of tasks still scale with problem size?
- Can task number be reduced without limiting flexibility?
- Are the engineering costs reasonable?

Do we have sufficient execution efficiency?

Mapping

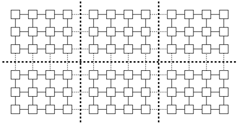
We need a strategy for mapping tasks to processors (cores).

- Only a problem for systems with distributed memory or shared memory with non-uniform memory access.
  - On multi-core processors and SMP systems, the automatic placement of tasks to cores by the OS suffices.
- Conflicting goals:
  - Place tasks that are able to execute concurrently on different processors.
  - Place tasks that communicate frequently on the same processor.

The mapping problem is NP-complete, so we can in general only hope for good heuristics.

Types of Mapping

- Static mappings:
  - A fixed number of permanent tasks is mapped at program start to processors; this mapping does not change.

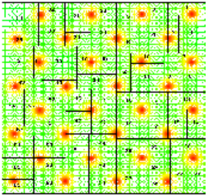


- Load balancing algorithms:
  - The assignment of permanent tasks to processors is adapted at runtime to keep processors equally busy.
- Task scheduling algorithms:
  - Many short-living tasks are created at runtime; a scheduler maps tasks to processors where they run until termination.

Static mapping is usually only sufficent for domain decomposition with structured communication.

Load Balancing: Recursive Bisection

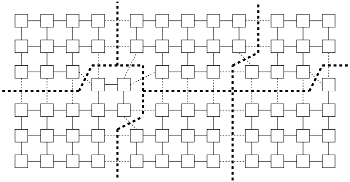
Recursively divide domain into partitions with equal costs.

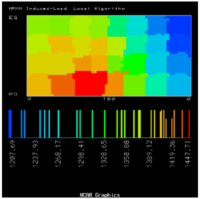
 

- Recursive coordinate bisection:
  - Recursively cut multi-dimensional grid at longest dimension.
- Unbalanced recursive bisection:
  - Choose among partitions the one with lowest aspect ratio.
- Recursive graph bisection:
  - Decompose graph according to distance from extremities.

Load Balancing: Local Algorithms

Compare load with that neighbor processors; transfer load if difference gets too big.

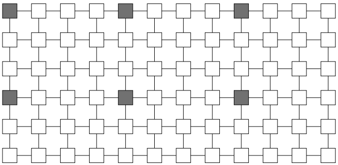




Use only local information and that of neighbor processors.

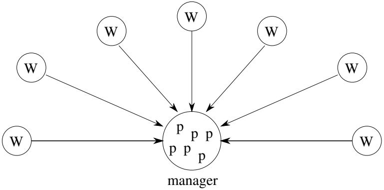
Load Balancing: Probabilistic/Cyclic Mapping

- Probabilistic mapping:
  - Map tasks to randomly selected processors.
  - If task number is much larger than processor number, every processor receives about the same amount of computation.
  - Generally leads to high communication.
- Cyclic mapping:
  - Map tasks to processors in a cyclic (scattered) mapping.
  - Each of P processors receives every P-th task in turn.
  - Similar to probabilistic mapping but more regular structure.



Task Scheduling

Maintain pool of tasks to which all new tasks are added.



- Manager/worker scheme:
  - Manager controls pool; idle workers ask manager for tasks.
- Hierarchical manager/worker scheme:
  - Subsets of workers with own submanagers and subpools.
  - Submanagers interact with manager (and each other).
- Decentralized schemes:
  - Each worker maintains its own task pool.
  - Idle workers request tasks from other workers.

Termination detection may become an issue.

Mapping Design Checklist

- If considering a program where tasks are only created at startup, have you also considered task scheduling?
- If considering task scheduling, have you also considered a program where tasks are only created at startup?
- If considering load-balancing, have you evaluated simpler alternatives such as probabilistic or cyclic mappings?
- If considering probabilistic or cyclic mappings, have you verified that task number is large enough to balance load?
- If considering task scheduling, have you verified that the manager does not become a bottleneck?

Do we have sufficient processor utilization?

General Recommendations

- Be sure to parallelize the actual hotspots of a program.
  - First you must understand where computation time is spent.
- Consider alternatives.
  - Do not just implement the first scheme that comes to mind.
- Remember scalability.
  - You may get more cores available than originally thought.
- But also consider the coding effort.
  - A simple solution may be sufficient as a starting point.
- And do not forget the application context.
  - The parallel code must be integrated into a bigger system.

Ultimately, determining the most efficient parallelization strategy for a given problem may require multiple iterations of performance debugging and optimizing/rewriting the code.