Parallel Program Design

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

A Methodical Approach

Design proceeds in various stages

- Machine-independent issues are considered early in the design.
  - concurrency
  - scalability

  Design abstract algorithm with these qualities.
- Machine-specific aspects are deferred to the end of the design.
  - locality
  - other performance-related issues

  Consider efficient execution on real architectures.

Design Stages

1. Partitioning

   Decompose computation and data into small tasks; find opportunities for parallelism.

2. Communication

   Determine communication required to coordinate task execution and define appropriate communication structures.

3. Aggolomeration

   Evaluate task and communication structures with respect to performance requirements; combine tasks into larger tasks.

4. Mapping

   Assign each task to a processor such that processor utilization is maximized and communication costs are minimized.

(See Foster, Figure 2.1)

The Partitioning Stage

Expose opportunities for parallel execution; determine a fine-grained decomposition of the problem.

Focus of partitioning may be on

- Data $\Rightarrow$ domain decomposition

  Partition data and then work out how to associate computation with data.

- Computation $\Rightarrow$ functional decomposition

  Partition computation into tasks and then associate data to tasks.

Complementary approaches; they may be applied to different parts of a problem and/or yield alternative algorithms for the same problem.

Domain Decomposition

Decompose data associated with a problem.

- Divide data into pieces of approximately equal size.
- Partition computation by associating each operation with the data on which it operates.
- Set of tasks = (data, operations)
- If operation requires data from several tasks, communication is required.
- Example: decomposition of 3D grid into one task for each grid point.

Typically for problems with large central data structures.
(See Foster, Figure 2.2)

Functional Decomposition

Decompose computation that is to be performed.

- Determine set of disjoint tasks.
- Determine data requirements of each task.
- If requirements overlap, communication is required.

Applie to problems without central data structures or to different parts of a problem.
(See Foster, Figure 2.3)

Partition Design Checklist

1. There should be at least an order of magnitude more tasks than processors.

   Flexibility for further design stages.

2. Avoid redundant computations and data.

   Otherwise algorithm may not be scalable for large problems.

3. Tasks should be of comparable size.

   Otherwise, load balancing may become difficult.

4. Task number should scale with problem size.

   Otherwise, algorithm is restricted to small problems on small machines.

5. Reconsider alternative partitions.

   Check both domain and functional decompositions.

The Communication Stage

Specify flow of information between tasks

- Communication structure (“channels”)

  Connections between tasks that require data (“consumers”) with those that possess those data (“producers”).

- Communication contents (“messages”)

Optimizations

- Avoid unnecessary channels and communication operations
- Distribute communication operations over many tasks
- Organize communication such that concurrent execution is possible.

Conceptual structure of a parallel program.

Communication Structure

Determining communication requirements

- Functional decomposition: simple

  Data flow between tasks.

- Domain decomposition: complex

  Operations may require data from several tasks; organizing communication in an efficient way can be difficult.

Communication Types

Essential characteristics of communication

- **Local vs. global**

  Communication with small set of tasks ("neighbors") or with many other tasks.

- **Structured vs. unstructured**

  Task and neighbors form a regular structure (tree, grid, ...) or arbitrary graphs.

- **Static vs. dynamic**

  Identity of communication partners known at compile time and does not change or depends runtime data and may be variable.

- **Synchronous vs. asynchronous**

  Producers/consumers execute in a coordinated fashion cooperating in data transfer or consumer may require data without cooperation of producer.

Local Communication

Example: Jacobi finite difference method

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

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

Easy parallelization, but many iterations.

Improvement: Gauss-Seidel strategy

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

Elements are updated in a particular order.

(See Foster, Figures 2.4 and 2.5)

Global Communication

Example: parallel reduction operation

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

Initial: central manager approach

- Time complexity $O(N)$
- Centralized algorithm

  Computation and communication operations are not distributed.
- Sequential algorithm

  Computation and communication operations cannot proceed concurrently.

Communication structure must be reorganized.
(See Foster, Figure 2.6)

Global Communication

1. Distribute communication/computation

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

   Concurrency only for multiple summations!

2. Uncover concurrency: divide&conquer

   $\Sigma_{i=0}^{2^n-1} X_i = \Sigma_{i=0}^{2^{n-1}-1} X_i + \Sigma_{i=2^{n-1}}^{2^n-1} X_i$

   Concurrency within one summation!

```
divide&conquer:
  if base_case then
    solve_problem
  else
    partition problem into L and R
    solve subproblem L with d&c
    solve subproblem R with d&c
    combine solutions of L and R
```

(See Foster, Figures 2.7 and 2.8)

Unstructured/Dynamic Communication

Communication patterns

- Complex structures,
- Depending on input data,
- Changing over time.

Agglomeration and mapping difficult.

Example: Finite Element Method
(See Foster, Figure 2.9)

Asynchronous Communication

- Producers do not know when consumers require data.
- Consumers must explicitly request data from producers.
- Typically for set of tasks that operate on shared data structure.

How to manage data structures that are arbitrarily accessed by many tasks?

Shared Data Structures

1. Distribute data structure among computation tasks.

   Each task has to regularly poll during computation for pending requests.

2. Distribute data structure among additional server tasks.

   Only purpose of these tasks is to serve read/write requests.

3. Use a shared-memory system.

   Direct read/write possible; synchronization of tasks required.

Performance characteristics depends on machine.

Communication Design Checklist

1. Have all tasks the same number of communication operations?

   Otherwise non-scalability; try to distribute communication.

2. Does each task communicate with a small set of neighbors?

   Otherwise non-scalability; try to reorganize algorithm.

3. Can communication operations proceed concurrently?

   Otherwise non-scalability; try to use divide-and-conquer.

4. Can computations in different tasks proceed concurrently?

   Otherwise non-scalability; try to reorder communication and computation.

The Agglomeration Stage

After partitioning and communication:

- Large number of small tasks,
- Large amount of communication.

Algorithm not efficient on real computers.

Combine tasks into larger tasks

1. Increase task granularity.

   Reduce communication costs.

2. Retain design flexibility.

   Scalability and mapping decisions.

3. Reduce engineering costs.

   Development overhead.

(See Foster, Figure 2.11)

Increasing Granularity

- Surface to volume effects

  Communication $\sim$ surface of subdomain; computation $\sim$ subdomain volume.

  Example: 2D Grid

  - "surface" scales with problem size,
  - "volume" scales with problem size squared.

  Communication/computation ratio decreases as task size increases.

- Replicating communication

  Trade off replicated computation for reduced communication.

  Example: replicated summation

- Avoiding Communication

  Agglomerate tasks that cannot execute concurrently.

(See Foster, Figures 2.13, 2.14, 2.15)

Preserving Flexibility

- Agglomeration must not limit scalability,
- No unnecessary limits on number of tasks,
- Task number may increase when problem and/or machine grows.

Task number should be an order of magnitude larger than processor number!

> If several tasks are mapped to the same processor, the processor does not become idle when a task is blocked in communication.

Overlapping computation and communication!

Reducing Software Engineering Costs

Efficiency and flexibility are not the only design criteria!

Consider also development costs:

- Avoid extensive code changes

  In a multidimensional grid code, it may be advantageous to avoid partitioning in one dimension, if then existing routines can be reused without change.

- Consider application context
  - Parallel algorithm is part of larger program,
  - Different data structures/decompositions may be required in context,
  - Restructuring of data may be required.

Agglomeration Design Checklist

1. Has agglomeration reduced communication costs by increasing locality?
2. If agglomeration replicates computation, do benefits outweigh costs?
3. If agglomeration replicates data, is scalability not compromised?
4. Have tasks similar computation and communication costs?
5. Does the task number still scale with problem size?
6. Is ther still sufficient concurrency for future target computers?
7. Can task number be further reduced without introducing load imbalances?
8. How far must the code be modified?

The Mapping Stage

Where does each task execute?

Only for distributed memory computers

On shared memory systems, the operating system efficiently schedules executable tasks to available processors.

- Place tasks that are able to execute concurrently on *different* processors.

  Enhance concurrency.

- Place tasks that communicate frequently on the *same* processor.

  Increase locality.

Mapping problem is NP-complete!

Mapping Strategies

- Static mapping

  Fixed number of equal-sized tasks and structured local and global communication (simple domain decomposition).

  (See Foster, Figure 2.16)

- Load balancing

  Variable amounts of work per task or unstructured communication patterns (more complex domain decomposition).

- Dynamic load balancing

  Number of tasks or amount of computation or communication per task changes.

- Task scheduling algorithms

  Many short-lived tasks that coordinate with other tasks only at start and end of execution (functional decomposition).

Load Balancing Algorithms

- Recursive bisection

  Partition a domain recursively into subdomains of equal computational cost.

  Coordinate bisection, unbalanced bisection, graph bisection (see Foster, Plates 1 and 2).

- Local algorithms

  Only use information from neighbor processors.

  Compare load with that of neighbors; transfer load if difference exceeds treshold (see Foster, Figure 2.17 and Plate 3).

Load Balancing Algorithms

- Probabilistic methods

  Allocate tasks to randomly selected processors.

  Low cost, good load balancing if number of tasks is much higher than number of processors, but high communication.

- Cyclic mappings

  Allocate tasks to processors in a round-robin fashion.

  Similar to probabilistic (see Foster, Figure 2.18).

Task Scheduling Algorithms

- Applicable when functional decomposition yields many small tasks.
- Task pool is maintained into which tasks are placed for allocation to processors.
- Tasks become data structures representing "subproblems" to be solved by set of workers (processors).

Which strategy is applied to allocate problems to workers?

Conflicting goals:

- Independent operation (to reduce communication costs),
- Global knowledge of computation state (to improve load balance).

Task Scheduling Algorithms

Strategies to allocate problems to workers

- Manager/worker
  - Central manager task distributes problems,
  - Idle worker asks manager for work,
  - Workers send new tasks to manager.
- Hierarchical manager/worker
  - Subsets of workers with own submanager,
  - Submanagers communicate with manager to balance load among worker sets.
- Decentralized schemes
  - Separate task pool on each processor,
  - Idle processors request work from other processors.

(See Foster, Figure 2.19)

Mapping Design Checklist

1. SPMD algorithm: consider dynamic task creation

   Simpler algorithm.

2. Dynamic task creation: consider SPMD algorithm

   Greater control over scheduling of computation and communication.

3. Centralized load-balancing: veryify manager does not become bottleneck.
4. Dynamic load-balancing: consider probabilistic/cyclic mappings.
5. Probablisitic/cyclic methods: verify that number of tasks is large enough.