Parallel Patterns

1 Problem Domain

1.1 Task Decomposition

Summary:

It specifies the decomposition of a problem into independent tasks to be processed in parallel.

Context:

Applications where the decomposition of the computation into tasks is most intuitive.

Forces:

- Number
- Granularity
- Parallelism

Solution

- Identify as many tasks as possible: functional decomposition, independent loop iterations, processing of recursive data structures
- Select suited granularity for efficiency and load balancing.
- Identify dependencies among tasks.
- Build groups of parallel tasks.
- Hoist dependences out of groups.
- Classify data as task local or shared.
1.2 Data Decomposition

Summary:

It specifies the decomposition of the data into blocks that can be processed in parallel.

Context:

Applications where the computationally intensive part is organized around the manipulation of large data structures.

Same operations are applied to different parts of the data structure.

Forces:

- Size of the data domain
- Computational load per element
- Access pattern
- Parallelism

Solution

- Identify a decomposition scheme that leads to good load balance and reduced communication.
- Identify if different decompositions are required in phases of the computation.
- Common schemes: multidimensional decompositions; block, cyclic, blockcyclic schemes; irregular decompositions.
- Identify local and remote data.
- Be aware of load balance and communication.
2 Algorithmic Structure

2.1 Organize by tasks

2.1.1 Independent Task Execution

Summary:

It specifies how independent tasks can be managed efficiently.

Context:

Applications with task decomposition.

Forces:

- Number of tasks in the group.
- Computational load
- Data accesses

Solution

- Identify whether there is a load balancing problem requiring special care when scheduling the tasks.
- Identify whether there are scheduling requirements based on previous and subsequent groups with respect to the data access.
- Make sure to synchronize dependent task groups.

2.1.2 Aggregation of Tasks

Summary:

Task can be too fine granular so that the scheduling overhead is dominant. In that case the aggregation of tasks can lead to more efficient execution.

Context:
Task decomposition. Too fine granular tasks.

Forces:

- Scheduling overhead
- Task execution time.

Solution

- Aggregate tasks based on an aggregation parameter that can later be fine tuned.
- Consider trade off with load balancing.

2.1.3 Recursive Tasks

Summary:

This patterns describes how recursive task problems can be efficiently executed.

Context:

Application with divide-and-conquer based task decomposition. This is an application with a special dependence pattern.

Forces:

- Execution time for split and merge. If it is high, parallelization will not be scalable.
- Dynamic reduction of granularity.
- Data movement for subproblems.
• Task graph is data dependent.

Solution

• For each solve create a new task.
• Threshold defines when to stop creating new tasks and to fall back to the sequential algorithm.
  • If subproblems are of same size, stop if number of active subtasks is the same as the number of PEs.
  • If problem is irregular, create finer tasks and use dynamic scheduling.

2.1.4 Static Task Scheduling

Summary:
The pattern specifies the approach to schedule task in a task group efficiently if no dynamic load balancing is required.

Context:
Task groups with equal tasks.

Forces:

• Communication of input and output data.
• Cache locality.

Solution

• Select one of the following scheduling strategies:
  • Block: no load balancing but good locality.
  • cyclic: load balance for triangular computations.
  • blockcyclic: combines both aspects.
  • irregular: allows to precompute a schedule that leads to load balance.
• Use a blocking parameter that can be fine-tuned.

2.1.5 Dynamic Task Scheduling

Summary:
It specifies how dynamic load balancing of tasks can be implemented efficiently.

Context:
Execution of task with varying and unpredictable size.

Forces:
• Scheduling overhead
• Task granularity

Solution
• Distribute new task once a task was finished.
• Combine tasks into blocks that are distributed to reduce the scheduling overhead.
• First distribute larger tasks to reduce the imbalance at the end of the execution.
  • Based on knowledge about the task size.
  • or by continuously reducing the block parameter (guided scheduling)

2.1.6 Dataflow Task Scheduling

Summary:
It specifies how tasks with dependences can be efficiently scheduled.
Context:
Task group with dependences.

Forces:
- Scheduling overhead

Solution
- Implement a scheme where tasks are dynamically assigned when the input data are available.
- Create the task graph and check for tasks that are ready.
- Aggregate ready tasks to reduce the scheduling overhead.

2.1.7 Synchronized Task Scheduling

Summary:
This pattern describes how tasks that have dependences and interact during their execution can be executed.

Context:
Tasks that communicate during their execution.

Forces:
- Available hardware parallelism.

Solution
- Use dataflow task scheduling.
Make sure that all ready tasks are active to ensure progress by using an appropriate number of UE or application level threads.

2.2 Organize by Data Decomposition

2.2.1 Static distribution Pattern

Summary:

This pattern specifies the implementation of a static data decomposition.

Context:

Applications where neither the load nor the access behavior changes dynamically.

Forces:

- computational load per element
- access pattern for elements

Solution

- Compute the distribution of the underlying data once.
- Determine remote data accessed based on the owner-computes rule.

2.2.2 Redistribution Pattern

Summary:

This pattern outlines the algorithmic structure for problems requiring dynamic redistribution.

Context:

Applications following data decomposition with dynamic load and/or access pattern changes.
Forces:

- Overhead for communication.
- Overhead for redistribution of data.

Solution

- Determine at which points in the algorithm a new decomposition is appropriate.
- Due to the overhead for redistribution a certain imbalance or higher communication overhead can be accepted.
- Check for imbalance after a configurable number of iterations of the progress loop. The parameter determines the tradeoff between the redistribution overhead and the losses due to imbalance and communication.
- Execute computation according to the owner-computes rule.
2.2.3 Irregular Distribution Pattern

Summary:

This pattern describes how the computation of the data distribution can be done for irregular access patterns.

Context:

Dataparallel applications with an irregular, input-dependent access pattern.

Forces:
• Memory and execution overhead for determining the decomposition.
• Size of the data structure and the connectivity.

Solution

• Collect the access pattern between elements, e.g. as an adjacency matrix.
• Apply a partitioning algorithm to compute a decomposition with equal load and short boundaries.
• Either perform this computation offline or online.
• Computation might need to be parallelized to handle memory and time complexity.
• Provide send/receive lists based on the decomposition to exchange data.

2.2.4 Regular Distribution Pattern

2.2.5 Oversubscription Pattern

Summary:

This pattern outlines the implementation of applications where the computational load per data points cannot be taken into account in the decomposition step.

Context:

Applications where the computational load per grid point varies and is not predictable.

Forces:

• Degree of variation
• Overhead for oversubscription of UEs
• Communication overhead due to more and finer blocks.

Solution

• Perform a decomposition into more parts then UEs.
Let each UE executed multiple blocks so that the load variation is distributed with some probability.

2.3 Organize by data flow

2.3.1 Pipeline Pattern

Summary:

Pipelining can be applied in applications that apply a sequence of operations to each element in a data set.

Context:

- Typical for streaming applications.
- Execution of a sequence of calculations on a sequence of data elements.
- Calculations on different elements are independent.
- Dependences specify the order of the tasks per data element.

Forces:

- Target platform can include special-purpose hardware for some operations.
- Limited amount of parallelism.
- Computational load of the tasks

Solution

- Assign each task to a PE and design a mechanism to forward data from one task to another.
- Termination can be implemented by appending a termination data record to the data set.
- Speedup limited by number of stages and the slowest stage.
- During filling and draining, the pipeline does not work as efficient.
- If cost of sending individual elements is high, aggregation is important.
- In a shared memory environment, shared queues might be appropriate to implement buffers between the stages.
2.3.2 Event-Based Coordination Pattern

Summary:

This pattern describes an approach to implement concurrent execution of a group of semi-independent tasks interacting in an irregular fashion.

Context:

Many examples can be found in the field of discrete-event simulation, i.e., simulation of a physical system consisting of a collection of objects whose interaction is represented by a sequence of discrete events.

Car washing station

Other example: Combine sequential program components that interact in a irregular way.

Forces:

- Solution should make it simple to express ordering constraints and enable as many activities as possible to be performed concurrently.

Solution

- Data flow abstraction implemented as events that are generated by a task and processed by another.
Asynchronous communication of events to enable the generating task to continue without waiting for the receiving task to consume the event.

Ordering constraints may make it necessary to process events in a different order from the order in which they are generated, e.g., wait for another event or wait until simulation time is reached when the event should be processed.

Load balancing is a difficult issue due to the irregular structure.

3 Implementation Concepts

3.1 Program Structures

3.1.1 SPMD

Summary:

This pattern describes the implementation of parallel algorithms via a single and replicated program.

Context:

Data parallel algorithms where all UEs execute the same operations on distributed memory systems.

Forces:

- Slightly different operations on the PEs.
- Complexity of implementation
- Efficiency
Solution:

- Single source code implementing the following elements
  - Initialize
  - Obtain unique identifier
  - Control different behavior either by branching or using the UE identifier in loop index calculations.
  - Distribute data
    - store local chunks only
    - or share or replicate data and modify only parts
  - Change accesses to local and remote
  - Communicate remote data
  - Adapt work distribution
  - Finalize

3.1.2 Master/Worker

Summary:

The master/worker pattern allows to dynamically load balance tasks.

Context:

- The workload of tasks are highly variable.
- The program structure doesn't map onto simple loops.
- Heterogeneous collection of PEs with dynamically varying capabilities
- Distributed memory system.

Forces:

- Varying capabilities of the PEs
- Size of the tasks due to scheduling overhead
Solution:

- **Master**
  
  Initialize
  
  Create a bag of tasks
  
  Launch workers
  
  distribute initial tasks to workers
  
  repeat until all tasks are executed
  
  wait for a result
  
  send next tasks
  
  Terminate

- **Worker**
  
  Initialize
  
  Repeat fetch and compute tasks until no more tasks are available.

- **Variations**
  
  - Detecting completion taking into account that tasks can create new tasks.
  
  - Master distributes tasks
  
  - Master may join the workers
  
  - Usage of a distributed task queue
  
  - Can provide a modest level of fault tolerance

### 3.1.3 Loop Parallelism

**Summary:**

This pattern describes the parallel implementation in the form of individual parallel loops.
Context:

- Loop based applications with task decomposition
- Large existing codes with incremental parallelization
- Shared memory systems

Forces:

- Memory utilization
- Global synchronization
- Overhead for starting and terminating parallel loops.

Solution:

- Transform loops into parallel loops
- Classify data into shared and private
- Insert synchronization to satisfy dependences.
- Select loop scheduling strategy to
  - Optimize load balancing
  - Limit the bandwidth requirements in the data path
  - Optimize for NUMA character

3.1.4 Fork/join

Summary:

Parallel execution can be implemented by dynamically starting and terminating new UEs for tasks.

Context:

Divide-and-conquer applications.
Forces:

- One-to-one mapping of tasks to UE might be favorable.
- System limit for the number of UEs
- Overhead for the creation and destruction of UEs

Solution:

- Create new UE for each task
- Use a UE pool that stores UEs that are currently not used.

3.2 Data Structures

3.2.1 Shared Data

1.1.1.1 Protected Data Structures

Summary:

This pattern specifies how shared data structures written by tasks can be protected by synchronization.

Context:

Task parallel codes with concurrent access to shared data structures, i.e., creating a list of values.

Forces:

- Duration of the operation
- Frequency of the operation
- Probability of conflicts
- Synchronization overhead
Solution:

- Protect data structure by mutual exclusion.
- If available use transactions and transactional memory
- Reduction operations: reduce frequency via local copies.

1.1.1.2 Lock-free Data Structures

Summary:

In contrast to protecting shared data structures via synchronization, lock-free data structures can be used. Lock-free data structures are based on atomic operations instead of locks to remove inconsistencies.

Context:

Task parallel codes with concurrent access to shared data structures, e.g., creating a list of values.

Forces:

- Duration of the operation
- Frequency of the operation
- Efficiency requirements

Solution:

- For many data structures, such as lists, lock-free operations have been developed.
1.1.1.3 Centralized Shared Queue

Summary:

Shared queues are required to implemented task scheduling. This pattern outlines the centralized approach.

Context:

Applications that require to distribute tasks dynamically according to their computational load. Low scalability requirements, i.e., small number or UEs.

Forces:

- Number of tasks
- Frequency of queue access
- Overhead for synchronization

Solution:

- Use a centralized queue into which new tasks can be inserted and UEs can retrieve tasks when they are idle.
- Centralized queue need to be protected.
- Special care needs to be taken due to the synchronization overhead.

1.1.1.4 Distributed Queue and work stealing

Summary:

The distributed queue pattern realizes a scalable task queue.

Context:

Task-based applications requiring dynamic load balancing. High number of UEs competing for access to the queue.
Forces:

- Dynamic creation of large number of tasks.
- Number of UEs

Solution:

- Create a number of task queues, e.g. one task queue per socket.
- New tasks go into the own task queue.
- UEs fetch tasks from their own task queue.
- If the own task queue is empty, a bunch of new tasks is stolen from other nodes.

### 3.2.2 Distributed Array

**1.1.1.5 Distributed Array with overlaps**

Summary:

Distributed arrays with overlaps allow to efficiently implement data accesses for stencil-based applications.

Context:

Applications with data decomposition and stencil-like data accesses.

Distributed memory systems.

Forces:

- Width of the stencil and the distribution scheme.
Solution:

- Add a shadow area to the local array of the distributed array.
- Fill in copies by communication
- Access the copies with normal unmodified array references.

1.1.1.6 Distributed Array with cache

Summary:

Distributed arrays with cache are an efficient implementation for data partitioning in codes with arbitrary remote accesses.

Context:

Application with data decomposition and accesses to arbitrary elements.

Forces:

- Communication overhead
- Frequency of accesses

Solution:
- Use an abstract data type that combines an array for the local elements with a cache for remote elements.
- Create copies of remote array elements if needed and cache the value.
- Flush the cache if new copies are required.
- All accesses to elements go through the access functions.

4 Implementation Mechanisms

4.1 MPI

4.1.1 Non-blocking Communication

Summary:

The usage of non-blocking communication allows to overlap communication and computation and reduces the performance hit due to communication.
Context:

MPI applications

Forces:

- Overhead
- Potential for overlap of communication and computation.

Solution:

- Replace blocking with non-blocking communication.
- Overlap communication with computation, may be by splitting a loop into the local computation and computation accessing remote data.

4.1.2 Virtual Topologies

Summary:

The virtual topology concept in MPI facilitates the implementation of data parallel applications and allows to optimize the mapping of UE to PEs.

Context:

- Data parallel applications
- Regular and irregular access patterns

Forces:

- Required communication pattern
- Mapping optimization in the MPI library
Solution:

- Use cartesian and graph topologies to express communication patterns.
- Initialize variables for the sender and receiver ranks based on the virtual topology.
- Allow rank reordering to improve the mapping of MPI processes to cores.

### 4.1.3 Single-sided Communication

**Summary:**

Single-sided communication reduces the overhead for message management in MPI. This pattern describes how to replace send/receives with single-sided communication.

**Context:**

- MPI programs
- Frequent communication of small messages.
- Significant MPI overhead

**Forces:**

- Potential advantage from the transformation.
- Implementation overhead
- Increased complexity

**Solution:**

- Replace two-sided communication with single-sided communication.
4.2 OpenMP

4.2.1 Atomic Synchronization

Summary:

Protection of shared data structures with locks or critical sections is expensive. If applicable, atomic operations can be used to achieve the same result.

Context:

- Shared memory programs
- OpenMP
- Shared data structures with locks or critical sections
- Significant competition for the lock
- Machine support for atomic
- Computation full fills the requirements for atomic

Forces:

- Code refactoring overhead

Solution:

- Replace the synchronization with atomic

4.2.2 First Touch Optimization

Summary:

Distributing the first touch of shared data structures according to the access pattern in the parallel part of the application leads to the appropriate distribution of the data structure across the node memories.

Context:
• Shared memory code
• NUMA architecture
• Little data locality for the data structure

Forces:

• Refactoring effort

Solution:

• Find the initialization of the data structure.
• May be it is an allocation with calloc which initializes it to 0.
• Create a parallel region with an appropriate work sharing construct. Touch the data structure with the same task distribution as in the computational kernel.

4.3 Fault tolerance

4.3.1 Regular Checkpointing

Summary:

During the execution of long running parallel application faults of the system might lead to a crash of the applications. Store the application status in regular intervals to be able to restart from the last checkpoint.

Context:

• Long running applications

Forces:

• Overhead for creating checkpoints
• Size of checkpoints
Solution:

- Implement a routine that stores the local state to a file. Carefully select the required data structures.
- Use MPI IO to efficiently write the file in collective operations. Individual files can also be used but this approach is not scalable.
- Provide a routine to read a previous state and to reinitialize the computation.
- Provide a parameter that controls the frequency.
- Insert the creation of the check point into the progress loop

4.3.2 Adaptive Checkpointing

Problem:

Manual checkpoint at regular intervals does not adapt to the overheads and the MTB of the system.

Context:

- Long running applications

Forces:

- Size of the checkpoint data and IO speed.
- MTB of the system

Solution:

- Carefully choose the checkpointing interval according to overhead and the MTB. While the MTB of the machine is fixed not application dependent, the overhead for storing the data definitely is and might be different from run to run depending on the data size and the IO system utilization.