Emulations, Scheduling and Patterns

May 11th, 2009

Content

- Emulations among architectures
- Task scheduling problem
- Scheduling algorithms
- Load balancing
- Patterns:
  - task decomposition,
  - data decomposition,
  - group tasks,
  - order tasks,
  - data sharing,
  - design evaluation

Emulations Among Architectures

- Reasons:
  - desire to quickly develop algorithms for a new architecture without expending the significant resources that would be required for native algorithm development.
  - develop algorithms for architectures that are easier to program (e.g., shared-memory or PRAM) and then have them run on machines that are realizable, easily accessible, or affordable.
- The developed algorithm then serves as a “source code” that can be quickly “compiled” to run on any given architecture via emulation.
- Emulation results are sometimes used for purposes other than practical porting of software.
- For example, we know that the hypercube is a powerful architecture and can execute many algorithms efficiently. Thus, one way to show that a new architecture is useful or efficient, without a need for developing a large set of algorithms for it, is to show that it can emulate the hypercube efficiently.
- If architecture A emulates architecture B with $O(f(p))$ slowdown and B in turn emulates C with $O(g(p))$ slowdown (assuming, for simplicity, that they all contain p processors), then A can emulate C with $O(f(p) \times g(p))$ slowdown.
- Problem solved with graph theory: embedding one graph in another one.

Task scheduling problem

- *Given a task system characterizing a parallel computation, determine how the tasks can be assigned to processing resources (scheduled on them) to satisfy certain optimality criteria*
- The *task system* is usually defined in the form of a directed graph, with nodes specifying computational tasks and links corresponding to data dependencies or communications
- *Optimality criteria* may include:
  - minimizing execution time,
  - maximizing resource utilization,
  - minimizing interprocessor communication,
  - meeting deadlines, or
  - a combination of these factors.
- *Task parameters:*
  - Execution or running time: the worst case, average case, or the probability distribution
  - Creation: a fixed set of tasks at compile time, or a probability distribution for task creation times
  - Relationship with other tasks: criticality, priority order, and/or data dependencies.
  - Start or end time
    - A task’s *release time* is the time before which the task should not be executed.
    - A hard or soft deadline may be associated with each task.
      - A *hard deadline* is specified when the results of a task become worthless if not obtained by a certain time.
      - A *soft deadline* may penalize late results but does not render them totally worthless.
- *Resources or processors* on which tasks are to be scheduled are typically characterized by their ability to execute certain classes of tasks and by their performance or speed.

Characteristics of scheduling algs.

1. Preemption.
   - Nonpreemptive scheduling: a task must run to completion once started,
   - Preemptive scheduling: execution of a task may be suspended to accommodate a more critical or higher-priority task.
   - In practice, preemption involves some overhead for storing the state of the partially completed task and for continually checking the task queue for the arrival of higher-priority tasks.
     - However, this overhead is ignored in most scheduling algorithms for reasons of tractability.

2. Granularity.
   - Fine-grain, medium-grain, or coarse-grain scheduling problems deal with tasks of various complexities, from simple multiply–add calculations to large, complex program segments, perhaps consisting of thousands of instructions.
   - Fine-grain scheduling is often incorporated into the alg, as otherwise the overhead would be prohibitive.
   - Medium- and coarse-grain scheduling are not fundamentally different, except that with a larger number of medium-grain tasks to be scheduled, the computational complexity of the scheduling algorithm can become an issue, especially with on-line or run-time scheduling.

Complexity

- Most interesting task scheduling problems are NP-complete.
  - This difficulty has given rise to research results on many special cases that lend themselves to analytical solutions and to a great many heuristic procedures that work fine under appropriate circumstances or with tuning of their decision parameters.
- Polynomial time optimal scheduling algorithms exist only for very limited classes of scheduling problems.
  - Examples include scheduling tree-structured task graphs with any number of processors and scheduling arbitrary graphs of unit-time tasks on two processors
- Most practical scheduling problems are solved by applying heuristic algorithms
  - Example: list scheduling

List scheduling

- A priority level is assigned to each task.
- A task list is then constructed in which the tasks appear in priority order, with some tasks tagged as being ready for execution (initially, only tasks having no prerequisite are tagged).
- With each processor that becomes available for executing tasks, the highest-priority tagged task is removed from the list and assigned to the processor.
- If q processors become available simultaneously, then up to q tasks can be scheduled at once.
- As tasks complete their executions, thus satisfying the prerequisites of other tasks, such tasks are tagged and become ready for execution.
- When all processors are identical, the schedulers differ only in their priority assignment schemes.
- In the case of unit-time tasks, tagging of new tasks can be done right after scheduling is completed for the current step.
- With different, but deterministic, running times, tagging can be done by attaching a time stamp, rather than a binary indicator, with each task that will become ready in a known future time.

- Example of possible priority assignment scheme for list scheduling.
  - Consider the simple case of unit-time tasks and ignore scheduling and communication overheads.
  - First the depth $T_\infty$of the task graph, which is an indicator of its minimum possible execution time, is determin.
  - Then take T_∞ as goal for the total running time T_p with p processors and determine the latest possible time step in which each task can be scheduled if our goal is to be met.
  - This is done by "layering" the nodes beginning with the output node.
  - The priority of tasks is then assigned in the order of the latest possible times.
  - Ties can be broken in various ways, e.g. give priority to a task with a larger number of descendants

Other examples of scheduling algs.

- Sophisticated scheduling algorithms may take communication delays, deadlines, release times, resource requirements, processor capabilities, and other factors into account.
- Because most scheduling algorithms do not guarantee optimal running times anyway, a balance must be struck between the complexity of the scheduler and its performance in terms of the total schedule length achieved.
- The simplicity of the scheduler is particularly important with on-line or run-time scheduling, where the scheduling algorithm must run on the parallel system itself, thus using time and other resources like the tasks being scheduled.
- With *off-line* or compile-time scheduling, the running time of the scheduler is less of an issue.
- If timing parameters, such as task deadlines and release times, are considered in making scheduling decisions, we have a *real-time* scheduling problem or scheduler.
  - Example scheduling strategies for real-time tasks include “nearest-deadline first” or “least-laxity first,” where the *laxity* of a task with deadline d and execution time e at current time t is defined as $d – t – e$.
- When the possibility of failure for processors and other resources is considered in task assignment or reassignment, we have *fault-tolerant* scheduling.

Distributed scheduling & Load balancing

- Distributed:
  - initially distribute the tasks among the available processors, based on some criteria,
  - let each processor do its own internal scheduling (ordering the execution of its set of tasks) according to interdependencies of tasks and the results received from other processors
  - The advantage of this approach is that most scheduling decisions are performed in a distributed manner.
  - A possible drawback is that the results may be far from optimal.
  - If such a scheme is combined with a method for redistributing the tasks when the initial distribution proves to be inappropriate, good results may be achieved.
- When load balancing?
  - Suppose the tasks are distributed to processors in such a way that the total expected running times of the tasks assigned to each processor are roughly the same.
  - Because task running times are not constants, a processor may run out of things to do before other processors complete their assigned tasks.
  - Also, some processors may remain idle for long periods of time as they wait for prerequisite tasks on other processors to be executed.
- In these cases, a *load balancing* policy may be applied in an attempt to make the load distribution more uniform.
  - switch as yet unexecuted tasks from an overloaded processor to a less loaded one.
  - load balancing can be initiated by an idle or lightly loaded processor (receiver-initiated) or by an overburdened processor (sender-initiated).

Load balance overhead

- Load balancing may involve a great deal of overhead that reduces the potential gains.
- If moving a task from one processor to another means copying a large program with huge amounts of data and then updating various status tables to indicate the new location of the task, then communication overhead is significant
  - load balancing may not be worth its cost.
- If the tasks belong to a standard set of tasks, each of which is invoked with a small set of parameters (data) and with copies already available locally to every processor, then moving the tasks may involve only a small broadcast message to pass the parameters and update the system status tables.
  - the load balancing overhead will be minimal.
- In circuit-switched networks that use wormhole routing, the load balancing problem can be formulated as a network flow problem
  - The excess (deficiency) of work load at some nodes may be viewed as flow sources (sinks) and the requirement is to allow the excess work load to “flow” from sources to sinks via paths that are, to the extent possible, disjoint and thus free from conflicts.

Self-scheduling systems

- The ultimate in automatic load-balancing is a self-scheduling system
  - Tries to keep all processing resources running at maximum efficiency.
  - There may be a central location to which processors refer for work and where they return their results.
  - An idle processor requests that it be assigned new work by sending a message to this central supervisor and in return receives one or more tasks to perform
  - This works nicely for tasks with small contexts and/or relatively long running times.
- A hardware-level implementation of such a self- scheduling scheme, known as *dataflow computing,* has a long history
  - A dataflow computation is characterized by a dataflow graph, which is very similar to the task graphs, but may contain decision elements and loops.
  - If an edge is restricted to carry no more than one token: a *static dataflow* system.
  - If multiple tagged tokens can appear on the edges and are “consumed” after matching their tags, we have a *dynamic dataflow* system
    - allows computations to be pipelined but implies greater overhead as a result of the requirement for matching of the token tags
  - Hardware implementation of dataflow systems with fine-grain computations (one or a few machine instructions per node), though feasible, has proven impractical.
  - When each node or task is a computation thread consisting of longer sequences of machine instructions, then the activation overhead is less serious and the concept becomes quite practical.

Patterns in the Design Space

- **Decomposition Patterns.**
  - Task Decomposition
  - Data Decomposition
- **Dependency Analysis Patterns.**
  - Group Tasks,
  - Order Tasks,
  - Data Sharing
- **Design Evaluation Pattern.**
  - important because it often happens that the best design is not found on the first attempt, and the earlier design flaws are identified, the easier they are to correct.
- In general, working through the patterns in this space is an iterativeprocess.

Task vs. data decomposition

- The **task decomposition** dimension views the problem as a stream of instructions that can be broken into sequences called *tasks* that can execute simultaneously.
  - the operations that make up the task should be largely independent of the operations taking place inside other tasks.
- The **data decomposition** dimension focuses on the data required by the tasks and how it can be decomposed into distinct chunks.
  - The computation associated with the data chunks will only be efficient if the data chunks can be operated upon relatively independently.
- Viewing the problem decomposition in terms of two distinct dimensions is somewhat artificial.
  - A task decomposition implies a data decomposition and vice versa;
  - The two decompositions are really different facets of the same fundamental decomposition.
  - We divide them into separate dimensions, however, because a problem decomposition usually proceeds most naturally by emphasizing one dimension of the decomposition over the other.
  - By making them distinct, we make this design emphasis explicit and easier for the designer to understand.

Ex.1 to be studied: Medical imaging

- PET (Positron Emission Tomography) scans provide an important diagnostic tool by allowing physicians to observe how a radioactive substance propagates through a patient's body.
  - The images formed from the distribution of emitted radiation are of low resolution, due in part to the scattering of the radiation as it passes through the body.
  - It is also difficult to reason from the absolute radiation intensities, because different pathways through the body attenuate the radiation differently.
- To solve this problem, models of how radiation propagates through the body are used to correct the images.
- A common approach is to build a Monte Carlo model.
  - Randomly selected points within the body are assumed to emit radiation (usually a gamma ray), and the trajectory of each ray is followed.
  - As a particle (ray) passes through the body, it is attenuated by the different organs it traverses, continuing until the particle leaves the body and hits a camera model, thereby defining a full trajectory.
  - To create a statistically significant simulation, thousands or millions of trajectories are followed.
- This problem can be parallelized in two ways.
  - Because each trajectory is independent, it is possible to parallelize the application by associating each trajectory with a task.
  - Another approach would be to partition the body into sections and assign different sections to different processing elements.

Ex. 2 to be studied: matrix multiplication

- If *T, A*, and *C* are square matrices of order *N*, matrix multiplication is defined such that each element of the resulting matrix *C* is where the subscripts denote particular elements of the matrices.
- In other words, the element of the product matrix *C* in row *i* and column *j* is the dot product of the *i*th row of *T* and the *j*th column of *A*.
- Hence, computing each of the N^2 elements of *C* requires *N* multiplications and *N*-1 additions, making the overall complexity of matrix multiplication O(N^3).
- There are many ways to parallelize a matrix multiplication operation.
  - It can be parallelized using either a task-based decomposition or a data-based decomposition [see previous lectures]

Ex. 3 to be studied: N-body problem

- Molecular dynamics is used to simulate the motions of a large molecular system.
  - Example, molecular dynamics simulations show how a large protein moves around and how differently shaped drugs might interact with the protein.
  - Molecular dynamics is extremely important in the pharmaceutical industry.
  - It is also a useful test problem for computer scientists working on parallel computing: It is straightforward to understand, relevant to science at large, and difficult to parallelize effectively.
- The basic idea is to treat a molecule as a large collection of balls connected by springs.
  - The balls represent the atoms in the molecule,
  - the springs represent the chemical bonds between the atoms.
- The molecular dynamics simulation itself is an explicit time-stepping process.
  - At each time step, the force on each atom is computed and then standard classical mechanics techniques are used to compute how the force moves the atoms.
  - This process is carried out repeatedly to step through time and compute a trajectory for the molecular system.
- The forces due to the chemical bonds (the "springs") are relatively simple to compute.
  - These correspond to the vibrations and rotations of the chemical bonds themselves.
  - These are short range forces that can be computed with knowledge of the handful of atoms that share chemical bonds.
  - The major difficulty arises because the atoms have partial electrical charges.
  - Hence, while atoms only interact with a small neighborhood of atoms through their chemical bonds, the electrical charges cause every atom to apply a force on every other atom.

  This is the famous *N*-body problem.

Ex. 3 to be studied: N-body problem

- On the order of N^2 terms must be computed to find the non-bonded forces.
- Because N is large (tens or hundreds of thousands) and the number of time steps in a simulation is huge (tens of thousands), the time required to compute these non-bonded forces dominates the computation.
- Several ways have been proposed to reduce the effort required to solve the problem
  - The simplest one: the cutoff method.
    - Even though each atom exerts a force on every other atom, this force decreases with the square of the distance between the atoms.
    - Hence, it should be possible to pick a distance beyond which the force contribution is so small that it can be ignored.
    - By ignoring the atoms that exceed this cutoff, the problem is reduced to one that scales as $O(N \times n)$, where n is the number of atoms within the cutoff volume, usually hundreds.
    - The computation is still huge, and it dominates the overall runtime for the simulation, but at least the problem is tractable.
- The primary data structures hold the atomic positions (atoms), the velocities of each atom (velocity), the forces exerted on each atom (forces), and lists of atoms within the cutoff distance of each atoms (neighbors).
- The program itself is a time stepping loop,
  - In which each iteration computes the short range force terms, updates the neighbor lists, and then finds the non-bonded forces.
  - After the force on each atom has been computed, a simple ordinary differential equation is solved to update the positions and velocities.
  - Physical properties based on atomic motions are then updated

Task Decomposition Pattern

- The key to an effective task decomposition is to
  - ensure that the tasks are sufficiently independent so that managing dependencies takes only a small fraction of the program's overall execution time.
  - ensure that the execution of the tasks can be evenly distributed among the ensemble of PEs (the load-balancing problem).
- Done by hand based on knowledge of the probl. & code required to solve it.
- Tasks can be found in many different places
  - In some cases, each task corresponds to a distinct call to a function.
    - Defining a task for each function call leads to what is sometimes called a functional decomposition.
  - Another place to find tasks is in distinct iterations of the loops within an algorithm.
    - If the iterations are independent and there are enough of them, then it might work well to base a task decomposition on mapping each iteration onto a task.
    - This style of task based decomposition leads to what are sometimes called loop splitting algorithms.
  - Tasks also play a key role in data-driven decompositions.
    - In this case, a large data structure is decomposed and multiple units of execution concurrently update different chunks of the data structure.
    - In this case, the tasks are those updates on individual chunks.

Task Decomposition - Examples

- Ex1 – medical imaging
  - It is natural to associate a task with each trajectory.
  - These tasks are particularly simple to manage concurrently because they are completely independent
  - The tasks need access to the model of the body
  - The body model can be extremely large.
  - Because it is a read-only model, this is no problem if there is an effective shared-memory system;
  - If the target platform is based on a distributed memory architecture, however, the body model will need to be replicated on each PE.
    - This can be very time consuming and can waste a great deal of memory.
    - if memory and/or network bandwidth is a limiting factor, a decomposition that focuses on the data might be more effective.
- Ex2 – matrix multiplication
  - Natural: the calculation of each element of the product matrix as a separate task
  - straightforward to implement in a shared memory environment.
  - Memory access time is slow compared to floating point arithmetic, so the bandwidth of the memory subsystem would limit the performance.
  - A better approach would be to design an algorithm that maximizes reuse of data loaded into a processor's caches.

Task Decomposition - Examples

- Ex3 – molecular dynamics
  - a natural task definition is the update required by each atom, which corresponds to a loop iteration in the sequential version.
  - After performing the task decomposition, one obtain the following tasks:
    - Tasks that find the vibrational forces on an atom
    - Tasks that find the rotational forces on an atom
    - Tasks that find the nonbonded forces on an atom
    - Tasks that update the position and velocity of an atom
    - A task to update the neighbor list for all the atoms
  - The key data structures are the neighbor list, the atomic coordinates, the atomic velocities, and the force vector.
  - Every iteration that updates the force vector needs the coordinates of a neighborhood of atoms.
  - The computation of non-bonded forces, however, potentially needs the coordinates of all the atoms,

Data Decomposition Pattern

- In shared-memory programming environments such as OpenMP, the data decomposition will frequently be implied by the task decomposition.
- In most cases, however, the decomposition will need to be done by hand, because the memory is physically distributed, because data dependencies are too complex without explicitly decomposing the data, or to achieve acceptable efficiency
- If a task-based decomposition has already been done, the data decomposition is driven by the needs of each task.
- If well-defined and distinct data can be associated with each task, the decomposition should be simple.
- When starting with a data decomposition, we need to look not at the tasks, but at the central data structures defining the problem and consider whether they can they be broken down into chunks that can be operated on concurrently.
- A few common examples include the following.
  - Array-based computations.
    - Concurrency can be defined in terms of updates of different segments of the array.
    - Array multidimensional can be decomposed in a variety of ways (rows, columns, or blocks).
  - Recursive data structures.
    - For example, decomposing the parallel update of a large tree data structure by decomposing the data structure into subtrees that can be updated concurrently.

Data decomposition - examples

- Ex1 – medical imaging
  - The body model is the large central data structure around which the computation can be organized.
  - The model is broken into segments, and one or more segments are associated with each processing element.
  - The body segments are only read, not written, during the trajectory computations, so there are no data dependencies created by the decomposition of the body model.
  - Each trajectory passing through the data segment defines a task.
  - The trajectories are initiated and propagated within a segment.
  - When a segment boundary is encountered, the trajectory must be passed between segments.
  - It is this transfer that defines the dependencies between data chunks.
- Ex2 – matrix multiplication
  - decompose the product matrix *C* into a set of row blocks (set of adjacent rows).
  - an even more effective approach that does not require the replication of the full A matrix is to decompose all three matrices into submatrices or blocks.

Data decomposition - examples

- Ex3 – molecular dynamics
  - The key data structures are
    - An array of atom coordinates, one element per atom
    - An array of atom velocities, one element per atom
    - An array of lists, one per atom, each defining the neighborhood of atoms within the cutoff distance of the atom
    - An array of forces on atoms, one element per atom
  - An element of the velocity array is used only by the task owning the atom.
    - This data does not need to be shared and can remain local to the task.
    - Every task, however, needs access to the full array of coordinates.
    - Thus, it will make sense to replicate this data in a distributed memory environment or share it among UEs in a shared memory environment.
  - More interesting is the array of forces.
    - From Newton's third law, the force from atom i on atom j is the negative of the force from atom j on atom i.
      - Exploit this symmetry: cut the amount of computation in half as we accumulate the force terms.
    - The values in the force array are not in the computation until the last steps in which the coordinates and velocities are updated.
    - Therefore, the approach used is to initialize the entire force array on each PE and have the tasks accumulate partial sums of the force terms into this array.
    - After all the partial force terms have completed, we sum all the PEs' arrays together to provide the final force array.

The Group Tasks Pattern

- Problem: How can the tasks that make up a problem be grouped to simplify the job of managing dependencies?
- the idea is to define groups of tasks that share constraints & simplify the probl of managing constraints by dealing with groups rather than individual tasks.
- Constraints among tasks fall into a few major categories.
  - The easiest dependency to understand is a temporal dependency—that is, a constraint on the order in which a collection of tasks executes.
    - If task A depends on the results of task B, for example, then task A must wait until task B completes before it can execute.
  - Another type of constraint: when a collection of tasks must run at the same time.
    - Ex: the original problem domain is divided into multiple regions that can be updated in parallel and the update of any given region requires information about the boundaries of its neighboring regions.
    - If all of the regions are not processed at the same time, the parallel program could stall or deadlock as some regions wait for data from inactive regions.
  - In some cases, tasks in a group are truly independent of each other.
    - Because it means they can execute in any order, including concurrently.
- The goal of this pattern is to group tasks based on these constraints, because
  - By grouping tasks, we simplify the establishment of partial orders between tasks, since ordering constraints can be applied to groups rather than to individual tasks.
  - Grouping tasks makes it easier to identify which tasks must execute concurrently.

Group Tasks - examples

- Ex – matrix multiplication
  - update of one element in *C*.
  - The memory organization of most modern computers, however, favors larger grained tasks such as updating a block of *C*.
  - Mathematically, this is equivalent to grouping the elementwise update tasks into group corresponding to blocks, and grouping the tasks this way is well suited to an optimum utilization of system memory.
- Ex – molecular dynamics
  - Tasks that find the vibrational forces on an atom
    - Tasks that find the rotational forces on an atom
    - Tasks that find the nonbonded forces on an atom
    - Tasks that update the position and velocity of an atom
    - A task to update the neighbor list for all the atoms
  - Consider how these can be grouped together.
    - Each item in the previous list corresponds to a high level operation in the original problem and defines a task group.
    - In each case the updates implied in the force functions are independent - the only dependency is the summation of the forces into a single force array.
    - The tasks in first two groups are independent but share the same constraints.
    - In both cases, coordinates for a small neighborhood of atoms are read and local contributions are made to the force array=> merge these into a single group for bonded interactions.
    - The other groups have distinct temporal or ordering constraints =>should not be merged.

The Order Tasks Pattern

- Problem: Given a way of decomposing a problem into tasks and a way of collecting these tasks into logically related groups, how must these groups of tasks be ordered to satisfy constraints among tasks?
- Constraints among tasks fall into a few major categories:
    - Temporal dependencies
    - Requirements that particular tasks must execute at the same time (ex: because each requires information that will be produced by the others).
    - Lack of constraint, that is, total independence.
- The purpose of this pattern is to help find and correctly account for dependencies resulting from constraints on the order of execution of a collection of tasks.
- There are two goals to be met when identifying ordering constraints among tasks and defining a partial order among task groups.
    - The ordering must be restrictive enough to satisfy all the constraints so that the resulting design is correct.
    - The ordering should not be more restrictive than it needs to be.
        - Overly constraining the solution limits design options and can impair program efficiency;
        - The fewer the constraints, the more flexibility you have to shift tasks around to balance the computational load among PEs.

Order Task – example: molecular dynamics

- Previously organized the tasks for this problem in the following groups:
  - A group of tasks to find the "bonded forces" (vibrational forces and rotational forces) on each atom
  - A group of tasks to find the nonbonded forces on each atom
  - A group of tasks to update the position and velocity of each atom
  - A task to update the neighbor list for all the atoms (which trivially constitutes a task group)
- Now consider ordering constraints between the groups.
  - The update of the atomic positions cannot occur until the force computation is complete.
  - The nonbonded forces cannot be computed until the neighbor list is updated.
- How these ordering constraints will be enforced: some sort of synchronization to ensure that they are strictly followed.

The Data Sharing Pattern

- Problem. Given a data and task decomposition for a problem, how is data shared among the tasks?
- The goal of this pattern is
    - to identify what data is shared among groups of tasks
    - determine how to manage access to shared data in a way that is both correct and efficient.
- The first step is to identify data that is shared among tasks.
    - This is most obvious when the decomposition is predominantly a data based decomposition.
- After the shared data has been identified, it needs to be analyzed to see how it is used.
- Shared data falls into one of the following three categories.
    - Read only
        - access to these values does not need to be protected.
        - on some distributed memory systs, it is worthwhile to replicate the read only data so each unit of execution has its own copy.
    - Effectively local.
        - The data is partitioned into subsets, each of which is accessed by only one of the tasks.
        - Ex: array shared among tasks so that its elements are effectively partitioned into sets of task local data.
        - If the subsets can be accessed independently it is not necessary to worry about protecting access
        - On distributed memory systems, such data would usually be distributed among UEs, with each UE having only the data needed by its tasks.
        - If necessary, the data can be recombined into a single data structure at the end of the computation.
    - Read-write.
        - any access to the data (read or write) must be protected with some type of exclusive access mechanism (locks, semaphores, etc.), which can be very expensive.
        - Ex: accumulate or multiple read/single write

Data Sharing – example: molecular dynamics

- The major shared data items are the following.

1. The atomic coordinates, used by each group.
   - These coordinates are treated as read-only data by the bonded force group, the nonbonded force group, and the neighborlist update group.
   - This data is read-write for the position update group.
   - The position update group executes alone after the other three groups are done
   - In the first three groups, leave accesses to the position data unprotected or even replicate it.
   - For the position update group, the position data belongs to the read write category, and access to this data will need to be controlled carefully.
2. The force array, used by each group except for the neighbor list update.
   - Is used as read-only data by the position update group and as accumulate data for the bonded and nonbonded force groups.
   - This array can be put in the accumulate category for the force groups and in the read-only category for the position update group.
   - The standard procedure for molecular dynamics simulations begins by initializing the force array as a local array on each UE.
   - Contributions to elements of the force array are then computed by each UE, with the precise terms computed being unpredictable because of the way the molecule folds in space.
   - After all the forces have been computed, the local arrays are reduced into a single array, a copy of which is place on each UE.
3. The neighbor list, shared between nonbonded force group & neighbor list update group.
   - Is essentially local data for the neighbor list update group and read only data for the nonbonded force computation.
   - The list can be managed in local storage on each UE.

The Design Evaluation Pattern

- Problem: Is the decomposition and dependency analysis so far good enough to move on to the next design space, or should the design be revisited?
- The design needs to be evaluated from three perspectives.
  1. Suitability for the target platform.
     - Issues such as no. processors and how data structures are shared will influence the efficiency of any design,
     - But the more the design depends on the target architecture, the less flexible it will be.
  2. Design quality.
     - Simplicity, flexibility, and efficiency are all desirable—but possibly conflicting—attributes.
  3. Preparation for the next phase of the design.
     - Are the tasks and dependencies regular or irregular
       - that is, are they similar in size, or do they vary?
     - Is the interaction between tasks synchronous or asynchronous
       - that is, do the interactions occur at regular intervals or highly variable or even random times
     - Are the tasks aggregated in an effective way?