CS 267: Applications of Parallel Computers

Dynamic Load Balancing

James Demmel
www.cs.berkeley.edu/~demmel/cs267_Spr14

Outline

• Motivation for Load Balancing
• Recall graph partitioning as static load balancing technique
• Overview of load balancing problems, as determined by
  • Task costs
  • Task dependencies
  • Locality needs
• Spectrum of solutions
  • Static - all information available before starting
  • Semi-Static - some info before starting
  • Dynamic - little or no info before starting
  • Or: how rapidly do costs/dependencies/locality needs change?
• Survey of solutions
  • How each one works
  • Theoretical bounds, if any
  • When to use it, tools

Sources of inefficiency in parallel codes

• Poor single processor performance
  • Typically in the memory system (recall matmul homework)
• Too much parallelism overhead
  • Thread creation, synchronization, communication
• Load imbalance
  • Different amounts of work across processors
  • Computation and communication
  • Different speeds (or available resources) for the processors
    • Possibly due to load on shared machine
    • Heterogeneous resources (eg CPU + GPU)
• How to recognize load imbalance
  • Time spent at synchronization is high and is uneven across processors, but not always so simple …

Measuring Load Imbalance

• Challenges:
  • Can be hard to separate from high synchronization overhead
  • Especially subtle if not bulk-synchronous
  • “Spin locks” can make synchronization look like useful work
  • Note that imbalance may change over phases
  • Insufficient parallelism always leads to load imbalance
  • Tools like IPM, TAU can help (acts.nersc.gov)
Review of Graph Partitioning – static case

- Partition G(N,E) so that
  - \( N = N_1 \cup \ldots \cup N_p \), with each \( N_i \sim N/p \)
  - As few edges connecting different \( N_i \) and \( N_k \) as possible
- If \( N = \{\text{tasks}\} \), each unit cost, edge \( e=(i,j) \) means task \( i \) has to communicate with task \( j \), then partitioning means
  - balancing the load, i.e. each \( N_i \sim N_i/p \)
  - minimizing communication volume
- Optimal graph partitioning is NP complete, so we use heuristics (see earlier lectures)
  - Spectral, Kernighan-Lin, Multilevel …
- Good software available
  - (Par)METIS, Scotch, Zoltan, …
- Speed of partitioner trades off with quality of partition
  - Better load balance costs more; may or may not be worth it
- Need to know tasks, communication pattern before starting
  - What if you don’t? Can redo partitioning, but not frequently

Load Balancing Overview

Load balancing differs with properties of the tasks

- Tasks costs
  - Do all tasks have equal costs?
  - If not, when are the costs known?
    - Before starting, when task created, or only when task ends
- Task dependencies
  - Can all tasks be run in any order (including parallel)?
  - If not, when are the dependencies known?
    - Before starting, when task created, or only when task ends
    - One task may prematurely end another task (eg search)
- Locality (may tradeoff with load balance)
  - Is it important for some tasks to be scheduled on the same processor (or nearby) to reduce communication cost?
  - When is the information about communication known?
- If properties known only when tasks end
  - Are statistics fixed, change slowly, change abruptly?

Task Cost Spectrum

Schedule a set of tasks under one of the following assumptions:

Easy: The tasks all have equal (unit) cost.

Harder: The tasks have different, but known, times.

Hardest: The task costs unknown until after execution.

\[
\begin{array}{cc}
\text{n items} & \text{p bins} \\
\text{branch-free loops} & \text{sparse matrix-vector multiply} \\
\text{GCM, circuits, search} & \text{dependence free loops} \\
\end{array}
\]

Task Dependency Spectrum

Schedule a graph of tasks under one of the following assumptions:

Easy: The tasks can execute in any order.

Harder: The tasks have a predictable structure.

Hardest: The structure changes dynamically (slowly or quickly)

\[
\begin{array}{cc}
\text{wave-front} & \text{out-tree} \\
\text{in-free balanced or unbalanced} & \text{general dag} \\
\text{search, sparse LU} & \text{matrix computations} \\
\text{dense, and some sparse, Cholesky) } & \\
\end{array}
\]
A key question is when certain information about the load balancing problem is known. Leads to a spectrum of solutions:

- **Static scheduling.** All information is available to scheduling algorithm, which runs before any real computation starts.
  - Off-line algorithms, e.g., graph partitioning, DAG scheduling
  - Still might use dynamic approach if too much information
- **Semi-static scheduling.** Information may be known at program startup, or the beginning of each timestep, or at other well-defined points. Offline algorithms may be used even though the problem is dynamic.
  - e.g., Kernighan-Lin, as in Zoltan
- **Dynamic scheduling.** Information is not known until mid-execution.
  - On-line algorithms – main topic today

**Dynamic Load Balancing**

- Motivation for dynamic load balancing
  - Search algorithms as driving example
- Centralized load balancing
  - Overview
  - Special case for schedule independent loop iterations
  - Makes most sense in shared memory environment
  - Hard to scale to large numbers of processors
- Distributed load balancing
  - Overview – randomization often used
  - Engineering
  - Theoretical results

**Search**

- Search problems are often:
  - Computationally expensive
  - Have very different parallelization strategies than physical simulations.
  - Require dynamic load balancing
- Examples:
  - Optimal layout of VLSI chips
  - Robot motion planning
  - Chess and other games (N-queens)
  - Speech processing
  - Constructing phylogeny tree from set of genes
Example Problem: Tree Search

- In Tree Search the tree unfolds dynamically
- May be a graph if there are common sub-problems along different paths
- Graphs unlike meshes which are precomputed and have no ordering constraints

Depth vs Breadth First Search (Review)

- DFS with Explicit Stack – little parallelism
  - Put root into Stack
  - Stack is data structure where items added to and removed from the top only
  - While Stack not empty
    - If node on top of Stack satisfies goal of search, return result, else
      - Mark node on top of Stack as "searched"
    - If top of Stack has an unsearched child, put child on top of Stack, else remove top of Stack
- BFS with Explicit Queue – lots of parallelism (depending on graph)
  - Put root into Queue
  - Queue is data structure where items added to end, removed from front
  - While Queue not empty
    - If node at front of Queue satisfies goal of search, return result, else
      - Mark node at front of Queue as "searched"
    - If node at front of Queue has any unsearched children, put them all at end of Queue
      - Remove node at front from Queue

Sequential Search Algorithms

- Depth-first search (DFS)
  - Simple backtracking
    - Search to bottom, backing up to last choice if necessary
  - Depth-first branch-and-bound
    - Keep track of best solution so far ("bound")
    - Cut off sub-trees that are guaranteed to be worse than bound
  - Iterative Deepening ("in between" DFS and BFS)
    - Choose a bound d on search depth, and use DFS up to depth d
    - If no solution is found, increase d and start again
    - Can use an estimate of cost-to-solution to get bound on d
- Breadth-first search (BFS)
  - Search all nodes at distance 1 from the root, then distance 2, and so on

Parallel Search

- Consider simple backtracking search
- Try static load balancing: spawn each new task on an idle processor, until all have a subtree

Load balance on 2 processors
Load balance on 4 processors

- We can and should do better than this …
Centralized Scheduling

- Keep a queue of task waiting to be done
  - May be done by manager task
  - Or a shared data structure protected by locks

Centralized Task Queue: Scheduling Loops

- When applied to loops, often called self scheduling:
  - Tasks may be range of loop indices to compute
  - Assumes independent iterations
  - Loop body has unpredictable time (branches) or the problem is not interesting
  - Originally designed for:
    - Scheduling loops by compiler (or runtime system)
    - Original paper by Tang and Yew, ICPP 1986

- Properties
  - Dynamic, online scheduling algorithm
  - Good for a small number of processors (centralized)
  - Special case of task graph – independent tasks, known at once

Variations on Self-Scheduling

- When applied to loops, often called self scheduling
  - Assume independent loop iterations, varying run times
  - Typically, don’t want to grab smallest unit of parallel work, i.e., a single loop iteration
    - Too much contention at shared queue
  - Instead, choose a chunk of tasks of size K.
    - If K is large, access overhead for task queue is small
    - If K is small, we are likely to have even finish times (load balance)
  - (at least) Four Variations:
    1. Use a fixed chunk size
    2. Guided self-scheduling
    3. Tapering
    4. Weighted Factoring

Variation 1/4: Fixed Chunk Size

- Kruskal and Weiss give a technique for computing the optimal chunk size (IEEE Trans. Software Eng., 1985)

- Requires a lot of information about the problem characteristics
  - e.g., task costs, number of tasks, cost of scheduling
  - Probability distribution of runtime of each task (same for all)
  - Assumes distribution is IFR = “Increasing Failure Rate”
    - For any t>0, P(X > x+t | X > x) is a decreasing function of x
  - \( K_{opt} = \frac{2^{2/3} \cdot \#tasks \cdot \text{time_to_access_queue}}{(\sigma \cdot p \cdot (\log p)^{2/3})^{2/3}} \)

- Not very useful in practice
  - Distribution must be known at loop startup time
Variation 2/4: Guided Self-Scheduling

- Idea: use larger chunks at the beginning to avoid excessive overhead and smaller chunks near the end to even out the finish times.
  - The chunk size $K_i$ at the $i^{th}$ access to the task pool is given by
    $$K_i = \text{ceiling}(R_i/p)$$
  - where $R_i$ is the total number of tasks remaining and
  - $p$ is the number of processors


Variation 3/4: Tapering

- Idea: the chunk size, $K_i$, is a function of not only the remaining work, but also the task cost variance
  - variance is estimated using history information
  - high variance $\Rightarrow$ small chunk size should be used
  - low variance $\Rightarrow$ larger chunks OK

  - Gives analysis (based on workload distribution)
  - Also gives experimental results -- tapering always works at least as well as GSS, although difference is often small

Variation 4/4: Weighted Factoring

- Idea: similar to self-scheduling, but divide task cost by computational power of requesting node

- Useful for heterogeneous systems
- Also useful for shared resource clusters, e.g., built using all the machines in a building
  - as with Tapering, historical information is used to predict future speed
  - “speed” may depend on the other loads currently on a given processor

- See Hummel, Schmit, Uma, and Wein, SPAA ‘96
  - includes experimental data and analysis

Summary: When is Self-Scheduling a Good Idea?

Useful when:

- A batch (or set) of tasks without dependencies
  - can also be used with dependencies, but most analysis has only been done for task sets without dependencies

- The cost of each task is unknown
- Locality is not important
- Shared memory machine, or at least number of processors is small -- centralization is OK
Cilk: A Language with Built-in Load balancing

A C language for programming dynamic multithreaded applications on shared-memory multiprocessors.

CILK (Leiserson et al) (supertech.lcs.mit.edu/cilk)
  • Created startup company called CilkArts
  • Acquired by Intel

Example applications:
  ● virus shell assembly
  ● graphics rendering
  ● n-body simulation
  ● heuristic search
  ● dense and sparse matrix computations
  ● friction-stir welding simulation
  ● artificial evolution

Cilk is a faithful extension of C. A Cilk program’s serial elision is always a legal implementation of Cilk semantics. Cilk provides no new data types.

Dynamic Multithreading

cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}

Algorithmic Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]

\[ T_\infty = \text{span}^* \]

 LOWER BOUNDS
  \[ T_P \geq T_1/P \]
  \[ T_P \geq T_\infty \]

*Also called critical-path length or computational depth.

Example: \( \text{fib}(4) \)

The computation dag unfolds dynamically.
**Speedup**

*Definition*: \( \frac{T_1}{T_P} = \text{speedup} \) on \( P \) processors.

If \( \frac{T_1}{T_P} = \Theta(P) \leq P \), we have linear speedup;

\( \frac{T_1}{T_P} = P \), we have perfect linear speedup;

\( \frac{T_1}{T_P} > P \), we have superlinear speedup,

which is not possible in our model, because of the lower bound \( T_P \geq \frac{T_1}{P} \).

\( \frac{T_1}{T_\infty} = \text{available parallelism} \)

\( = \text{the average amount of work per step along the span (critical path)} \).

---

**Greedy Scheduling**

*Idea*: Do as much as possible on every step.

*Definition*: A thread is *ready* if all its predecessors have executed.

*Complete step*

- \( \geq P \) threads ready.
- Run any \( P \).

*Incomplete step*

- \( < P \) threads ready.
- Run all of them.

---

**Cilk's Work-Stealing Scheduler**

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim's deque.

---

**Performance of Work-Stealing**

*Theorem*: Cilk’s work-stealing scheduler achieves an expected running time of

\[ T_P \leq \frac{T_1}{P} + O(T_\infty) \]

on \( P \) processors.

*Pseudoproof*. A processor is either *working* or *stealing*. The total time all processors spend working is \( T_1 \). Each steal has a \( 1/P \) chance of reducing the span by 1. Thus, the expected cost of all steals is \( O(P T_\infty) \). Since there are \( P \) processors, the expected time is

\[ \left( T_1 + O(P T_\infty) \right)/P = T_1/P + O(T_\infty) \].
Further analyses of Cilk’s Performance

- Space needed (for stacks) by \( P \) processors at most \( P \) times space needed by one processor
- Bounds on #cache misses caused by work stealing if each processor has local cache, single shared (slow) memory
- Bounds extended to hierarchical memories
- General conclusions:
  - Work stealing good idea if execution DAG not too deep, and sequential implementation would not generate too many cache misses

Space Bounds

**Theorem.** Let \( S_1 \) be the stack space required by a serial execution of a Cilk program. Then, the space required by a \( P \)-processor execution is at most \( S_P = P S_1 \).

**Proof** (by induction). The work-stealing algorithm maintains the busy-leaves property: every extant procedure frame with no extant descendents has a processor working on it.

DAG Scheduling software

- **QUARK** (U. Tennessee)
  - Library developed to support PLASMA for pipelining (“synchronization avoiding”) dense linear algebra
- **SMPss** (Barcelona)
  - Compiler based; Data usage expressed via pragmas; Proposal to be in OpenMP; Recently added GPU support
- **StarPU** (INRIA)
  - Library based; GPU support; Distributed data management; Codelets=tasks (map CPU, GPU versions)
- **DAGUE/DPLASMA** (MPI group work)
  - Needs a compact DAG representation; Distributed memory; Designed to be very, very scalable
- Other tools (e.g., fork-join graphs only)
  - Cilk, Intel Threaded Building Blocks (TBB); Microsoft CCR, …

Pipelining: Cholesky Inversion

**POTRF**+**TRTRI**+**LAUUM**: \( 25=(7t-3) \)
Cholesky Factorization alone: \( 3t-2 \)

Source: Julien Langou: ICL presentation 2011/02/04
Simplified QUARK architecture

Scheduling is done using a combination of task assignment to workers (via locality reuse, etc.) and work stealing.

Basic QUARK API

Setup QUARK data structures
QUARK_New [standalone] or
QUARK_Setup [part of external library]

For each kernel routine, insert into QUARK runtime
QUARK_Insert_Task (quark, function, task_flags, arg_size, arg_ptr, arg_flags, \ldots, \ldots, \ldots, 0);

When done, exit QUARK
QUARK_Delete [standalone] or
QUARK_Waitall [return to external library]

Other basic calls
QUARK_Barrier
QUARK_Cancel_Task
QUARK_Free (used after QUARK_Waitall)

Scalability of DAG Schedulers

- How many tasks are there in DAG for dense linear algebra operation on an n x n matrix with b x b blocks?
- \( O((n/b)^3) = 1M \), for \( n=10,000 \) and \( b = 100 \)
- Creating, scheduling entire DAG does not scale
- PLASMA: static scheduling of entire DAG
- QUARK: dynamic scheduling of “frontier” of DAG at any one time

Performance – 12 core

MKL is really good when there are a few cores
Performance – 24 core

Starting from standard layout, PLASMA static, PLASMA QUARK, MKL
Platform: 48 core (8 x 6-core) 2.8GHz Opteron; 128GB; peak 538 GFlop/s

PLASMA Static (24 threads)
QUARK (24 threads)
MKL (24 threads)

QUARK is pretty close to static PLASMA

Limitations: Future Work

- VERY sensitive to task size
  - For PLASMA, small tile sizes give bad performance, need NB around 180
  - Overhead kills performance for small tasks.
- Master handles serial task insertion
  - This is a hurdle for large scale scalability
  - Some work may be delegated in future versions
- Scalability
  - Largest tests are for 48 cores
  - Large scale scalability is untested
  - For ongoing work see icl.cs.utk.edu/iclprojects/

Performance – 48 core

PLASMA Static (48 threads)
QUARK (48 threads)
MKL (48 threads)

QUARK is approx 10% less than static; MKL scales up more slowly.

Trace: LU factorization

LU factorization (dgetrf) of N=5000 on 48 cores using dynamic QUARK runtime
Trace created using EZTrace and visualized using ViTE
Distributed Task Queues

• The obvious extension of task queue to distributed memory is:
  • a distributed task queue (or “bag”)
  • Idle processors can “pull” work, or busy processors “push” work

• When are these a good idea?
  • Distributed memory multiprocessors
  • Or, shared memory with significant synchronization overhead
  • Locality is not (very) important
  • Tasks may be:
    • known in advance, e.g., a bag of independent ones
    • dependencies exist, i.e., being computed on the fly
  • The costs of tasks is not known in advance

Distributed Dynamic Load Balancing

• Dynamic load balancing algorithms go by other names:
  • Work stealing, work crews, ...

• Basic idea, when applied to tree search:
  • Each processor performs search on disjoint part of tree
  • When finished, get work from a processor that is still busy
  • Requires asynchronous communication

How to Select a Donor/Acceptor Processor

• Three basic techniques:
  1. Asynchronous round robin
     • Each processor k, keeps a variable "target_k"
     • When a processor runs out of work, requests work from target_k
     • Set target_k = (target_k + 1) mod procs
  2. Global round robin
     • Proc 0 keeps a single variable "target"
     • When a processor needs work, gets target, requests work from target
     • Proc 0 sets target = (target + 1) mod procs
  3. Random polling/stealing
     • When a processor needs work, select a random processor and request work from it
  4. Random distribution of work
     • When a processor has too much work, select a random processor to take it

  • Repeat if no work is found

How to Split Work

• First parameter is number of tasks to give when asked
  • Related to the self-scheduling variations, but total number of tasks is now unknown

• Second question is which one(s)
  • Send tasks near the bottom of the stack (oldest)
  • Execute from the top (most recent)
  • May be able to do better with information about task costs
Theoretical Results (1)

Main result: Simple randomized algorithms are optimal with high probability

• Others show this for independent, equal sized tasks
  • “Throw n balls into n random bins”: $\Theta(\log n / \log \log n)$ in fullest bin
  • Throw d times and pick the emptiest bin: $\log \log n / \log d$ [Azar]
  • Extension to parallel throwing [Adler et al 95]
  • Shows $p \log p$ tasks leads to “good” balance

• Karp and Zhang show this for a tree of unit cost (equal size) tasks
  • Parent must be done before children
  • Tree unfolds at runtime
  • Task number/priorities not known a priori
  • Children “pushed” to random processors

Distributed Task Queue References

• Introduction to Parallel Computing by Kumar et al (text)
• Multipol library (See C.-P. Wen, UCB PhD, 1996.)
  • Part of Multipol (www.cs.berkeley.edu/projects/multipol)
  • Try to push tasks with high ratio of cost_to_compute/cost_to_push
    • Ex: for matmul, ratio = $2n^3 \text{cost(flop)} / 2n^2 \text{cost(send a word)}$
• Goldstein, Rogers, Grunwald, and others (independent work) have all shown
  • advantages of integrating into the language framework
  • very lightweight thread creation

Theoretical Results (2)

Main result: Simple randomized algorithms are optimal with high probability

• Blumofe and Leiserson [94] show this for a fixed task tree of variable cost tasks
  • their algorithm uses task pulling (stealing) instead of pushing, which is good for locality
  • I.e., when a processor becomes idle, it steals from a random processor
  • also have (loose) bounds on the total memory required
  • Used in Cilk
  • “better to receive than to give”

• Chakrabarti et al [94] show this for a dynamic tree of variable cost tasks
  • works for branch and bound, i.e. tree structure can depend on execution order
  • uses randomized pushing of tasks instead of pulling, so worse locality

Diffusion-Based Load Balancing

• In the randomized schemes, the machine is treated as fully-connected.
• Diffusion-based load balancing takes topology into account
  • Send some extra work to a few nearby processors
    • Average work with nearby neighbors
    • Analogy to diffusion (Jacobi for solving Poisson equation)
  • Locality properties better than choosing random processor
  • Load balancing somewhat slower than randomized
  • Cost of tasks must be known at creation time
  • No dependencies between tasks
• See Ghosh et al, SPAA96 for a second order diffusive load balancing algorithm
  • takes into account amount of work sent last time
  • avoids some oscillation of first order schemes
Diffusion-based load balancing
- The machine is modeled as a graph
- At each step, we compute the weight of task remaining on each processor
  - This is simply the number if they are unit cost tasks
- Each processor compares its weight with its neighbors and performs some averaging
  - Analysis using Markov chains
- See Ghosh et al, SPAA96 for a second order diffusive load balancing algorithm
  - Takes into account amount of work sent last time
  - Avoids some oscillation of first order schemes
- Note: locality is still not a major concern, although balancing with neighbors may be better than random

Measurement Based Load Balancing in Charm++
- Principle of persistence (A Heuristic)
  - Object communication patterns and computational loads tend to persist over time, so recent past good predictor of future
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
  - Only a heuristic, but applies on many applications
- Measurement based load balancing
  - Runtime system (in Charm++) schedules objects and mediates communication between them, so can measure load
  - Use the instrumented data-base periodically to make new decisions, and migrate objects accordingly
- Charm++ provides a suite of strategies, and plug-in capability for user-defined ones
  - Also, a meta-balancer for deciding how often to balance, and what type of strategy to use

Charm++
Load balancing based on Overdecomposition
- Context: “Iterative Applications”
  - Repeatedly execute similar set of tasks
- Idea: decompose work/data into chunks (chares in Charm++), and migrate chares for balancing loads
  - Chares can be split or merged, but typically less frequently (or unnecessary in many cases)
- How to predict the computational load and communication between objects?
  - Could rely on user-provided info, or based on simple metrics
    - E.g. number of elements
  - Alternative: principle of persistence
    - Statistics change slowly, can rebalance occasionally
- Software, documentation at charm.cs.uiuc.edu
  - Many applications: NAMD, LeanMD, OpenAtom, ChaNGa, ...

Periodic Load Balancing Strategies
- Many alternative strategies can use the same database
  - OCG: Object communication graph
    - Or simply #loads of each object, if communication unimportant
- Centralized strategies: collect data on one processor
  - Feasible on up to a few thousand cores, because number of objects is typically small (10-100 per core?)
  - Use Graph partitioners, or greedy strategies
  - Or refinement strategies: mandated to keep most objects on the same processors
    - Charm++ provides a suite of strategies, and plug-in capability for user-defined ones
    - Also, a meta-balancer for deciding how often to balance, and what type of strategy to use
### Load Balancing Steps

- **Regular Timesteps**: Detailed, aggressive Load Balancing
- **Instrumented Timesteps**: Refinement Load Balancing

### Periodic Load Balancing for Large machines

- Two Challenges:
  - Object communication graph cannot be brought to one processor
    - A solution: Hierarchical load balancer (next slide)
  - Interconnection topology must be taken into account
    - Limited bisection bandwidth (on Torus networks, for example)
    - Solution: topology-aware balancers (later slides)

### Charm++ Hierarchical Load Balancer Scheme

- **Refinement-based Load balancing**
- **OCG-based Load balancing**

### Topology-aware load balancing

- With wormhole routing, the number of hops a message takes has very little impact on transit time
  - But: On an unloaded network!
- But bandwidth is a problem
  - Especially on torus networks
  - More hops each message takes, more bandwidth they occupy
  - Leading to contention and consequent delays
- So, we should place communicating objects nearby
  - Many current systems are "in denial" (no topo-aware allocation)
    - Partly because some applications do well
  - Lot of research in the 1980’s
    - But not very relevant because of technological assumptions and topologies considered
  - Ex: Take advantage of physical proximity (domain decomp.)
**Topology aware load balancing (2/2)**

- Metric: Average dilation (equivalently, sum of hop-bytes)
- Object-based over-decomposition helps balancing
- When (almost) near-neighbor communication dominates
  - And geometric information available
  - Simplest case, but challenges: Aspect ratios, load variations,
  - Strategies: ORB, many heuristic placement strategies
    - (A. Bhatele Phd. Thesis)
  - Variation: A set of pairwise interactions (e.g. Molecular dynamics) among geometrically placed primary objects:
    - Strategy: place within the "brick" formed by the two primary obj
- When application has multiple phases:
  - Strategy: often blocking helps. Alternatively, optimize one phase (better than optimizing neither)
  - Example: OpenAtom for Quantum Chemistry

**Summary and Take-Home Messages**

- There is a fundamental trade-off between locality and load balance
- Many algorithms, papers, & software for load balancing
- Key to understanding how and what to use means understanding your application domain and their target
  - Shared vs. distributed memory machines
  - Dependencies among tasks, tasks cost, communication
  - Locality oblivious vs locality "encouraged" vs locality optimized
    - Computational intensity: ratio of computation to data movement cost
  - When you know information is key (static, semi, dynamic)
- Open question: will future architectures lead to so much load imbalance that even "regular" problems need dynamic balancing?