Dynamic Load Balancing

Rashid Kaleem and M Amber Hassaan

Scheduling for parallel processors

- Story so far
  - Machine model: PRAM
  - Program representation
    - control-flow graph
    - basic blocks are DAGs
      - nodes are tasks (arithmetic or memory ops)
      - weight on node = execution time of task
    - edges are dependencies
  - Schedule is a mapping of nodes to (Processors x Time):
    - which processor executes which node of the DAG at a given time

Recall: DAG scheduling

- Schedule work on basis of "length" and "area" of DAG.
- We saw
  - T_1 = Total Work (Area)
  - T_∞ = Critical path (Length)
- Given P processors, any schedule takes time $\geq \max(T_1/P, T_\infty)$
- Computing an optimal schedule is NP-complete
  - use heuristics like list-scheduling

Reality check

- PRAM model gave us fine-grain synchronization between processors for free
  - processors operate in lock-step
- As we saw last week, cores in real multicores do not operate in lock-step
  - synchronization is not free
  - therefore, using multiple cores to exploit instruction-level parallelism (ILP) within a basic block is a bad idea
- Solution:
  - raise the granularity of tasks so that cost of synchronization between tasks is amortized over more useful work
  - in practice, tasks are at the granularity of loop iterations or function invocations
  - let us study coarse-grain scheduling techniques

Lecture roadmap

- Work is not created dynamically
  - (e.g.) for-loops with no dependences between loop iterations
  - number of iterations is known before loop begins execution but work/iteration is unknown
  - ➔ structure of computation DAG is known before loop begins execution, but not weights on nodes
  - lots of work on this problem
- Work is created dynamically
  - (e.g.) worklist/workset based irregular programs and function invocation
    - even structure of computation DAG is unknown
  - three well-known techniques
    - work stealing
    - work sharing
    - diffusive load-balancing
- Locality-aware scheduling
  - techniques described above do not exploit locality
  - goal: co-scheduling of tasks and data
- Application-specific techniques
  - Barnes-Hut

For-loop iteration scheduling

- Consider for-loops with independent iterations
  - number of iterations is known just before loop begins execution
  - very simple computation DAG
    - nodes represent iterations
    - no edges because no dependences
- Goal:
  - assign iterations to processors so as to minimize execution time
- Problem:
  - if execution time of each iteration is known statically, we can use list scheduling
  - what if execution time of iteration cannot be determined until iteration is complete?
    - need some kind of dynamic scheduling

Important special cases

Constant Work

```
For (i=0 to N)
{
    doSomething();
}
```

Variable work

```
For (i=0 to N)
{
    if (checkSomething(i)   doSomething();
    else                    doSomethingElse();
}
```

Increasing Work

```
For (i=0;i<N;i++)
{
    SerialFor (j=1 to i)
      doSomething();
}
```

Decreasing Work

```
For (i=0 to N)
{
    SerialFor (j=1 to N-i)
      doSomething();
}
```

Dynamic loop scheduling strategies

- Model:
  - centralized scheduler hands out work
  - free processor asks scheduler for work
  - scheduler assigns it one or more iterations
  - when processor completes those iterations, it goes back to scheduler for more work
- Question: what policy should scheduler use to hand out iterations?
  - many policies have been studied in the literature

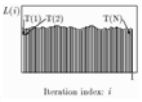
Loop scheduling policies

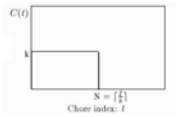
- Self Scheduling (SS)
  - One iteration at a time. If a processor is done with an iteration, it requests another iteration.
- Chunked SS (CSS)
  - Hand out `k' iterations at a time, when k is determined heuristically before loop begins execution
- Guided SS (GSS)
  - Start with larger "chunks", and decrease to smaller chunks with time. Chunk size = remaining work/processors.
- Trapezoidal SS (TSS)
  - GSS with linearly decreasing size function
  - TSS is parameterized by two parameters F,L
    - initial chunk size: F
    - final chunk size: L

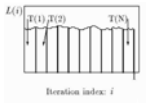
Scheduling policies(I)

- Chunk Size C(t) vs Time "chore" index
- Task size L(i) Vs Iteration index i

Self Scheduling



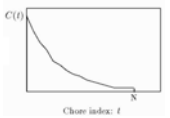


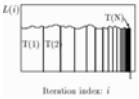


Chunked SS

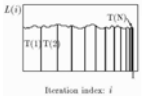
Scheduling policies (II)

- Chunk Size C(t) vs Time "chore" index
- Task size L(i) Vs Iteration index i





Guided SS



Trapezoidal SS

Problems

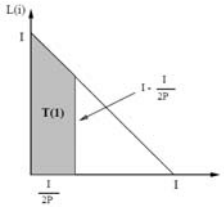
- SS and CSS are not adaptive, so they may perform poorly when work/iteration varies widely, such as with increasing and decreasing loads
- GSS would perform poorly on decreasing work load, especially if the initial chunk is the critical chunk.

Trapezoidal SS(F,L)

- Given the initial chunk size F and ending chunk size L, TSS can be adapted to SS, CSS or GSS.
  - SS = TSS(1,1)
  - CSS(k) = TSS(k,k)
  - GSS(k) ≈ TSS(Work/P, 1)
- So, TSS(F,L) can perform as others, but can we do better?

Optimal TSS(F,L)

- Consider TSS (Work/(2xP),1)
  - We divide the initial work into two, which we distribute amongst the P processors.
  - We linearly reduce the chunk size based on:
    - Delta = (F - L) / (N - 1)
    - Where N = (2 x Work) / (F + L)



Performance of TSS

- If F and L are determined statically, TSS performs as good as other self-sched schemes
- Larger initial chunk size reduces task assignment overhead similar to GSS
- GSS faces problem in decreasing workload since the initial allocation maybe the critical chunk. TSS handles this by ensuring half of work is divided in first allocation.
- Subsequent allocation reduce linearly, with all parameters pre-determined, hence efficiently.

Dynamic work creation

Dynamic work creation

- In some applications, doing some piece of work creates more work
- Examples
  - irregular applications like DMR
  - function invocations
- For these applications, the amount of work that needs to be handed out grows and shrinks dynamically
  - contrast with for-loops
- Need for dynamic load-balancing
  - processor that creates work may not be the best one to perform that work

Task Pools

- Basic mechanism: task pool (aka task queue)
  - all tasks are put in task pool
  - free processor goes to task pool and is assigned one or more tasks
  - if a processor creates new tasks, these are put into pool
- Variety of designs for task queues
  - Single task queue
    - Load balancing
      - guided scheduling
  - Split task queue
    - Load balancing
      - Passive approaches
        - » Work stealing
      - Active approaches
        - » Work sharing
        - » Diffusive load balancing

Single Task Queue

- A single task queue holds the “ready” tasks
- The task queue is shared among all threads
- Threads perform computation by:
  - Removing a task from the queue
  - Adding new tasks generated as a result of executing this task

Single Task Queue

- This scheme achieves load balancing
- No thread remains idle as long as the task queue is non-empty
- Note that the order in which the tasks are processed can matter
  - not all schedules finish the computation in same time

Single Task Queue: Issues

- The single shared queue becomes the point of contention
- The time spent to access the queue may be significant as compared to the computation itself
- Limits the scalability of the parallel application
- Locality is missing all together
  - Tasks that access same data may be executed on different processors
  - The shared task queue is all over the place

Single Task Queue: Guided Scheduling

- The work in the queue is chunked
- Initially the chunk size is big
  - Threads need to access the task queue less often
  - The ratio of computation to communication increases
- The chunk size towards the end of the queue is small
  - Ensures load balancing

Split Task Queues

- Let each thread have its own task queue
- The need to balance the work among threads arises
- Two kinds of load balancing schemes have been proposed
  - **Work Sharing**:
    - Threads with more work push work to threads with less work
    - A centralized scheduler balances the work between the threads
  - **Work Stealing**:
    - A thread that runs out of work tries to steal work from some other thread

Work Stealing

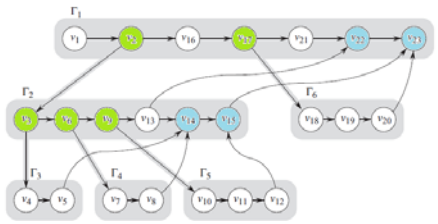
- Early implementations are by:
  - Burton and Sleep 1981
  - Halstead 1984 (Multi-Lisp)
- Leiserson and Blumofe 1994 gave theoretical bounds:
  - A work stealing scheduler produces an optimal schedule
  - Space required by execution is bounded
  - Communication is limited
    - $O(PT_\infty(1+n_d)S_{max})$

Strict Computations.

- Threads are sequence of unit time instructions.
- A thread can spawn, die, join.
  - A thread can only join to its parent thread.
  - A thread can only stall for its child thread.
- Each thread has an activation record.

Example.

- T1 is root thread. It spawns T2, T6 and Stalls for T2 at V22,V23 and T6 at V23.
- Any multithreaded Computation that can be executed in a depth first manner on a single processor can be converted to fully strict w/o changing the semantics.



Why fully Strict?

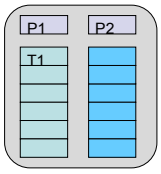
- A "realistic" model easier to analyze
- A fully strict computation can be executed depth-first by a single thread
- Hence we can always execute the "Leaf" Tasks in parallel.
  - Busy Leaves Property
- Consider any fully strict computation:
  - T_1 = total work
  - T_∞ = critical path length
- For a greedy schedule X,
  - $T(X) <= T_1/P + T_\infty$

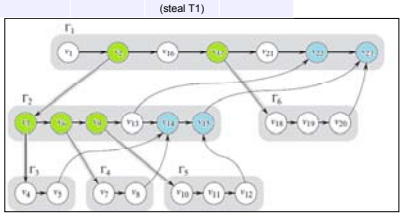
Randomized Work-stealing

- Processor has ready deque. For itself, this is a stack, others can "steal" from top.
  - A.Spawn(B)
    - Push A to bottom, start working on B.
  - A.Stall()
    - Check own "stack" for ready tasks. Else "steal" topmost from other random processor.
  - B.Die()
    - Same as Stall
  - A.Enable(B)
    - Push B onto bottom of stack.
- Initially, a processor starts with the "root" task, all other work queues are empty.

2-processors, at t=3

Time	1	2	3	4
P1	V1	V2 (spawn T2)	V3 (spawn T3)	V4
P2			V16 (steal T1)	V17

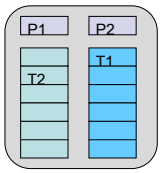


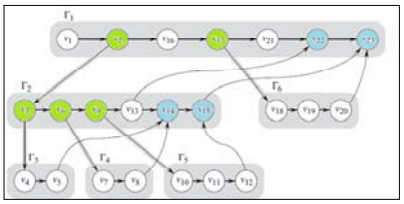


Work-list after t-3, P2 will "steal" T1 and begin executing V16.

2-processors, at t=5

Time	1	2	3	4	5	6
P1	V1	V2 (spawn T2)	V3 (spawn T3)	V4	V5 (die T3)	V6 (spawn T4)
P2			V16 (steal T1)	V17 (spawn T6)	V18	V19



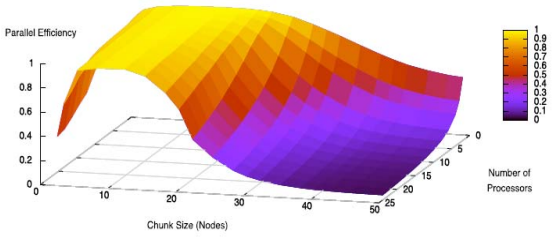


Work-list after t-5, P2 will work on T6 with T1 on its work-list and P1 is executing V5 with T2 on its work-list.

Work Stealing example: Unbalanced Tree Search

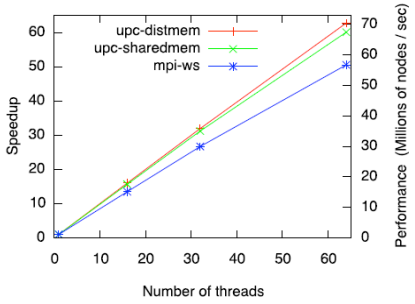
- The benchmark is synthetic
  - It involves counting the number of nodes in an unbalanced tree
  - No good way of partitioning the tree
- Olivier & Prins 2007 used work stealing for this benchmark
  - A thread traverses the tree Depth-First
  - Threads steal un-traversed sub-trees from a traversing thread
  - Work stealing gives good results

Unbalanced Tree Search



Variation of efficiency with work-steal chunk size
Results on a Tree of 4.1 million nodes on SGI Origin 2000

Unbalanced Tree Search



Speed up results for shared and distributed memory
Results on a Tree of 157 billion nodes on SGI Altix 3700

Work Stealing: Advantages

- Work Stealing algorithm can achieve optimal schedule for "strict" computations
- As long as threads are busy no need to steal
- The idle threads initiate the stealing
  - Busy ones keep working
- The scheme is distributed
- Known to give good results on Cilk and TBB

Work Stealing: Shortcomings

- Locality is not accounted for
  - Tasks using same data may be executing on different processors
  - Data gets moved around
- Still need mutual exclusion to access the local queues
  - Lock free designs have been proposed
  - Split the local queue into two parts:
    - Shared part for other threads to steal from
    - Local part for the owner thread to execute from
- Other Issues:
  - How to select a victim for stealing
  - How much to steal at a time

Work Sharing

- Proposed by Rudolph et al. in 1991
- Each thread has its local task queue
- A thread performs:
  - A computation
  - Followed by a possible balancing action
- A thread with L elements in its local queue performs a balancing action with probability 1/L
  - Processor with more work will perform less balancing actions

Work Sharing

- During a balancing action:
  - The thread picks a random partner thread
  - If the difference between the sizes of the local queues is greater than some threshold:
    - Local queues are balanced by migrating tasks
- Authors prove that load balancing is achieved.
- The scheme is distributed and asynchronous
- Load balancing operations are performed with the same frequency throughout.

Diffusive Load Balancing

- Proposed by Cybenko (1989)
- Main idea is:
  - Load can be thought of as a fluid or gas
    - Load is equal to number of tasks at a processor
  - The actual processor network is a graph
  - The communication links between processors have a bandwidth
    - Which determines the rate of fluid flow
- A processor sends load to its neighbors
  - If it has higher load than a neighbor
  - Amount of load transferred = (difference in load) x (rate of flow)
- The algorithm periodically iterates over all processors

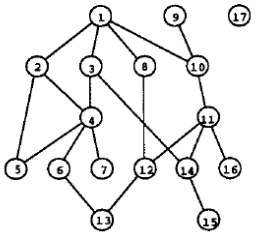
Diffusive Load Balancing

- Cybenko showed that for a D-dimensional hypercube the load balances in D+1 iterations
- Subramanian and Scherson 1994 show general bounds on the running time of load balancing algorithm
- The bounds on running time of actual parallel computation are not known

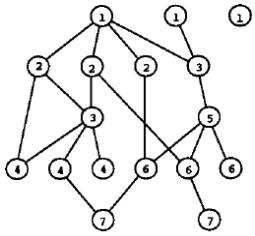
Parallel Depth First Scheduling

- Blelloch et al. in 1999 give a scheduling algorithm, which:
  - Assumes a centralized scheduler
  - Has optimal performance for strict computations
  - The space is bounded to 1+O(1) of sequential execution for strict computations
- Chen et al. in 2007 showed that Parallel Depth First has lower cache misses than Work Stealing algorithm

Parallel Depth First Scheduling



Depth First Schedule on a single thread



Parallel Depth First Schedule on p=3 threads

Parallel Depth First Scheduling

- The schedule follows the depth first schedule of a single thread
- Maintains a list of the ready nodes
- Tries to schedule the ready nodes on P threads
- When a node is scheduled it is replaced by its ready children in the list
  - Ready children are placed in the list left to right

Locality-aware techniques

Key idea

- None of the techniques described so far take locality into account
  - tasks are moved around without any consideration about where their data resides
- Ideally, a load-balancing technique would be locality-aware
- Key idea:
  - partition data structures
  - bind tasks to data structure partitions
  - move (task+data) to perform load-balancing

Partitioning

- Partition the Graph data structure into P partitions and assign to P threads
- Galois uses partitioning with lock coarsening:
  - The number of partitions is a multiple of number of threads
- Uniform partitioning of a graph does not guarantee uniform load balancing
  - E.g.: in DMR there may be different number of bad triangles in each partition
  - Bad triangles generated over the execution are not known
- Partitioning the graph for ordered algorithms is hard

Application-specific techniques

N-body Simulation: Barnes-Hut

- Singh et al.(1995) studied hierarchical N-body methods
  - Barnes-Hut, Fast Multipole, Radiosity
  - They proposed techniques for load balancing and locality based on insights into the algorithms
- We'll look at Barnes-Hut
- Iterate over time steps
  1. Subdivide space until at most one body per cell
     - Record this spatial hierarchy in an octree
  2. Compute mass and center of mass of each cell
  3. Compute force on bodies by traversing octree
     - Stop traversal path when encountering a leaf (body) or an internal node (cell) that is far enough away
  4. Update each body's position and velocity

Barnes-Hut: Load Balancing Insights

- Around 90% of the time is spent in force calculation
- The partitioning requirements are not same among all four phases
- Distribution of the particles determines:
  - Structure of the octree
  - Work per particle/cell
  - More work in denser parts of the domain
  - Dividing particles equally among processors does not balance loads
- Introduce a cost metric per particle
  - = number of interactions required for force computation
  - Cost per particle is not known before hand
  - The distribution of particles changes very slowly over time
    - Cost per particle does not change very often
  - Can be used for load balancing
    - Not good for position update phase

Barnes-Hut: Locality Insights

- Partition the actual 3D space
  - Use Orthogonal Recursive Bisection (ORB)
  - Divides the space into 2 subspaces recursively
    - Based on a cost function
    - The cost function here is the profiled cost per particle
  - Introduces a new data structure to manage
  - Number of processors should be a power of 2
- Partition the octree
  - Octree captures the spatial distribution of particles
  - Traverse the leaves left-to-right and sum the particle costs
  - Divide the leaves (and subtrees above them) based on cost
  - Leaves near each other in octree may not be near in 3D space
    - Needed for efficient tree building
    - Can be achieved by careful number of child cells

Barnes-Hut: Tree Partitioning

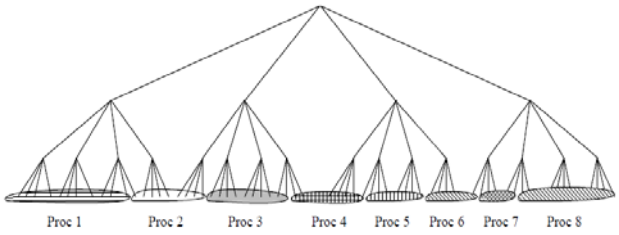
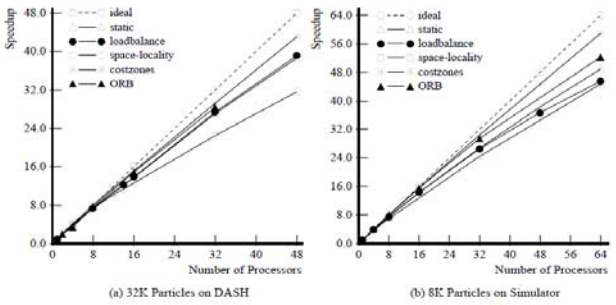
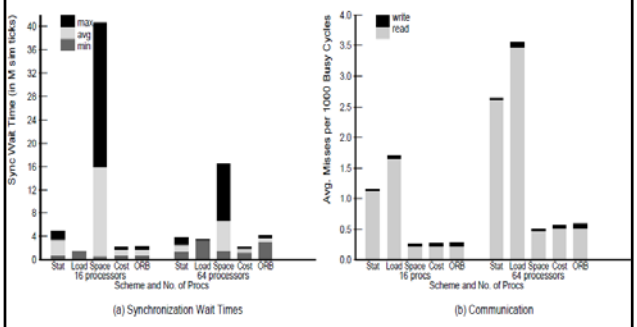


Figure 13: Tree partitioning in the *costzones* scheme.

Barnes-Hut: Results



Barnes-Hut: Simulation stats for 8K particles



Summary

- We reviewed some research on load balancing
- High-level idea
  - computation DAG is available statically: schedule at compile time
  - otherwise: some kind of dynamic scheduling/load-balancing is needed
- Almost all existing techniques ignore locality altogether
  - can you do better?
- Algorithm-specific insights may be necessary to achieve performance
  - can we use our science of parallel programming approach to design general-purpose mechanisms that achieve the same level of performance?