Parallel Algorithms

Lecture 7: Divide And Conquer

Ronald Veldema

Divide and Conquer (1)

- Take a problem 'X'
  - Subdivide in subproblems of equal type
    - Input, output domain, etc partitioning
    - Subproblems are independent of each other
  - Solve the subproblems
    - Recursively
    - sequentially
  - Combine subproblems' solutions
    - Wait for all subproblems to complete ?

Divide and Conquer (2)

- Also known as ``recursive decomposition'‘
- Top down reasoning:
  - split large jobs into smaller jobs until so small that it can be best solved sequentially.
  - merge results to solve the bigger task which in turn can be used to solve the next larger task.
- Take a recursive program
  - I.S.O. recursive calls, create parallel tasks
  - when result needed, wait for subtask to finish

Divide and Conquer (3)

- Communication only occurs when
  - Spawning a new task
  - Gathering/reducing results
- pro: no dependence in program on
  - the number of processors used
  - network topology
  - scheduling
- con: hard to tune algorithms by hand as hardly anything to tune.

Complexity….

- Say we have an D&C alg with a branchout of '2'
  - 2 subproblems are solved recursively
  - seq. time alg(N) = time alg(N/2) + time alg(N/2)
  - par. time alg(N) = time alg(smallest granularity)
    - Neglect time needed to traverse the binary tree

Example: fibonacci numbers (1)

$$\text{fib}(n) = \begin{cases} N & \text{, if } N < 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{, otherwise} \end{cases}$$

fib(n) = (n < 2) ? n : fib(n - 1) + fib(n - 2)

```
long sequential_fib(long n) {

        if(n < 2) return n;

        long x = fib(n-1);
        long y = fib(n-2);
        return x + y;
}
```

Example: fibonacci numbers (2)

fib(n) = (n < 2) ? n : fib(n - 1) + fib(n - 2)

```
long parallel_fib(long n) {

        if(n < 2) return n;

        long x = spawn task fib(n-1);
        long y = spawn task fib(n-2);
        <wait for sub tasks>
        return x + y;
 }
```

Example: Fibonacci Numbers (3)

- optimization: more sequential execution to make tasks more coarse grained.
  - How to chooze CUT_OFF ?

```
long parallel_fib(long n) {
            if(n < CUT_OFF) return sequential_fib(n);
            long x = spawn fib(n-1);
            long y = spawn fib(n-2);
            <wait for sub tasks>
            return x + y;
     }
```

When to stop dividing ?

- granularity
  - how much parallel ?
  - how much sequential ?
  - smaller granularity →
    - more communication but better load balancing...
    - More overhead
      - Sub task administration
      - Method call overheads

Example: Binary Search (1)

```
// A sorted table. The table is sorted on 'keys'
// using the KeyType.smaller_than() method
KeyType keys[];
ValueType values[];

ValueType binary_search(KeyType key, int start, int end)
{
    int middle := (start+end)/2;

    if (keys[middle].equals(key))
          return value[middle];
    else (keys[middle].smaller_than(key))
          return binary_search(X, start, middle-1);
    else
          return binary_search(key, middle+1, end);
}
```

Example: Binary Search (2)

- Any ideas on how to parallelize ?
  - Spawning recursive iterations won't help

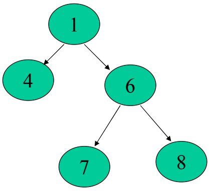
Example: Binary Search (3)

- When spawning job [0..n/2]
  - Spawn parallel jobs [0..n/4] and [n/4...n/2]

```
ValueType binary_search(KeyType key, int start, int end)
{
    int middle := (start+end)/2;

    if (keys[middle].equals(key))
          return value[middle];
    else (keys[middle].smaller_than(key))
          return parallel
                      binary_search(X, start, (start+middle-1)/2)
                      binary_search(X, (start+middle-1)/2, middle-1)
    else return parallel
                    binary_search(key, middle+1, (middle+1+end)/2);
                    binary_search(key, (middle+1+end)/2, end);
}
```

Example: Heap Sort (1)

- heap = balanced binary tree with value at any node smaller than that of its children



Example: Heap Sort (2)

```
Sequential_heap_sort(unordered set of numbers L)
{
        Heap H = new Heap( L );
        List X  = new List();
        while (H not empty)
                P = remove lowest number from H;
                // note: lowest number = top of the tree
                 X  += P;
        return X
}
```

Example: Heap Sort (3)

- Sequential Heap construction:

```
BinaryTree t = new BinaryTree();

For each number in list:
        find a good spot to insert element
```

Example: Heap Sort (4)

- Parallel_heap_sort(List L)
  - BalancedUnsortedBinaryTree T(L);
  - Heapify(T)

Example: Heap Sort (4)

```
Heapify(Tree T)
        if T = empty
                return;

        heapify(T.left);
        heapify(T.right);

        if (left.value <  T.value)
                swap_data(left, T);
        if (right.value < T.value)
                swap_data(right, T);
```

Example: Raytracer (1)

- For each pixel on the screen
  - Color[x,y] = render_pixel()
- Some pixels take longer to compute than others
- Pixels are independent

Example: Raytracer (2)

```
void render_screen(int witdth, int height) {
      for (int x=0;x<width;x++)
          for (int y=0;y<height;y++) {
                    Color color = render(x,y);
                    screen.plot_pixel(x, y, color);
          }
}
```

Example: Raytracer (3)

```
void render_screen_rectangle(rectangle r) {

    if (r.is_large()) {
        render_screen_rectangle(r.upper_left());
           render_screen_rectangle(r.upper_right());
           render_screen_rectangle(r.lower_left());
           render_screen_rectangle(r.lower_right());
           return;
    }

     for (int x=r.x; x<r.width;x++)
           for (int y=r.z; y<r.height;y++) {
                      Color color = render(x,y);
                   screen.plot_pixel(x, y, color);
        }
}
```

Dynamic Programming (DP) (0)

- Solve a problem recursively

  (Just like divide and conquer)
- Before recomputing a subproblem
  - See if you've already computed it
    - Return 'cached' answer
  - Recompute otherwise

Dynamic Programming (DP) (1)

- DP = recursion with memory
- where divide and conquer has no dependencies between tasks, dynamic programming assumes that there is
- bottom up reasoning I.S.O. top down reasoning.
  - solve smallest tasks first and record results.
  - try and solve next larger task and record result
  - goto 2 while not found solution.

Dynamic Programming (2)

- tasks that have already been computed are assumed to return the same value on another invocation
- solving the next bigger task can be viewed as combining sub-results using a function:
- result = f(s(i1), s(i2) ... s(in))
- monadic combining functions: contain 1 's' in f(..)
- polydic combining functions: contain >1 's' in f(..)
  - F(a(i1),b(i2), c(i2), etc)
- work can be represented in a directed graph.
- serial DP formulation: depend only on tasks that are one level deeper in graph
- DP requires communiction to store intermediate results where simple div&conquer requires none.

Dynamic Programming (3)

```
// parallel fib with dynamic programming
long cached_value[] = {-1, -1, -1, etc};

public long fib(long n) {
   if(n < 2) return n;
   long x = spawn dp_fib(n-1);
   long y = spawn dp_fib(n-2);
   sync();
   return x + y;
}

long dp_fib(long n) {
   if (cached_value[n] != -1)
        return cached_value[n];
   else
        cached_value[n] = fib(n);
     return valued_value[n];
}
```

Dynamic Programming (4)

- Knapsack problem:
  - given items 'i' of weight w(i) and profit p(i), try to fit as many objects in knapsack of capacity c while maximizing profit.
  - solution vector = (0/1)*, where 0 if not in sack, 1 if in sack.
  - F(i, x) = max profit solution for i objects in sack with x capacity
    - DP solution = 0 if x >= 0, i=0
    - -inf if x < 0, i=0
    - max(F(i-1,x), F(i-1,x-w(i)) + p(i)) if 1< i <=n
  - parallel:
    - F(i-1,x) can be computed on another machine in parallel to F(i-1,x-w(i))
    - results of computations on other machines can be broadcasted.

Knapsack problem (1)

Item	0	1	2	3	4	5	6	7	8
Value	5	0	3	3	5	6	3	2	4
Weight	2	4	5	4	4	3	5	2	2

Knapsack can hold 15 kilos

- try and put as many objects in the knapsack as possible

Knapsack problem (2)

```
Solution knapsack(int bitvector, int item) {
      if (weight(bitvector) > capacity) return INF;
      if (item == max_item) return new Solution(bitvector);
       // put item in knapsack
       int bit_vector1 = bitvector | (1<<item);
       // don't put item in knapsack
       int bit_vector2 = bitvector;

       Solution s1 = knapsack(bitvector1, item+1);
       Solution s2 = knapsack(bitvector2, item+1);

       if better(s1,s2) return s1
       else return s2;
}
```

Knapsack problem (3)

```
Solution knapsack(int bitvector, int item) {
      if (weight(bitvector) > capacity) return INF;
      if (item == max_item) return new Solution(bitvector);
       // put item in knapsack
       int bit_vector1 = bitvector | (1<<item);
       // don't put item in knapsack
       int bit_vector2 = bitvector;

       Solution s1 = spawn knapsack(bitvector1, item+1);
       Solution s2 = spawn knapsack(bitvector2, item+1);
       sync;
       if better(s1,s2) return s1
       else return s2;
}
```

Knapsack problem (4)

- Problems
  - If a better solution has been found, the algorithm will still look at knapsacks that are worse
  - These is no idea of a 'global optimum' until the very end
- Solution: maintain a global 'best capacity' value

Knapsack problem (5)

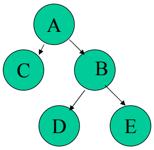
```
Solution knapsack(int bitvector, int item) {
      if (weight(bitvector) > best_found) return INF;
      best_found = min(best_found, weight(bitvector));
      if (item == max_item) return new Solution(bitvector);
       // put item in knapsack
       int bit_vector1 = bitvector | (1<<item);
       // don't put item in knapsack
       int bit_vector2 = bitvector;

       Solution s1 = spawn knapsack(bitvector1, item+1);
       Solution s2 = spawn knapsack(bitvector2, item+1);
       sync;
       if better(s1,s2) return s1
       else return s2;
}
```

Where is the bug ?

Dynamic programming (5)

- Problems:
  - Every processor needs concurrent read/write to the ‘cache’ of already found answers
    - Possible solutions:
      - Each processor maintains his own private cache
      - Partition cache & maintain a separate ‘lock’ for each partition
  - ‘cache’ can become very large if ALL answers are stored
    - Pushing out old return-values may help

Task Graphs (1)



- task A spawns B & C.
- task B spawns D & E.

- no incoming edges = start nodes
- no outgoing edges = end nodes
  - critical path = max start → end node.
  - critical path length = length(critical path)
- interaction graph = if data exchanges between tasks I & J, then edge between I & J.

Task Graphs (2)

- Divide & conquer: from parent to child (parameter & return value)
- Dynamic programming: from each internal node that may use or produce a cachable value

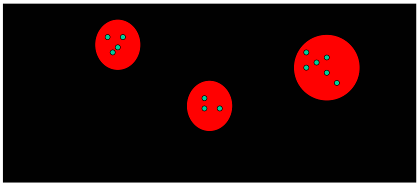
---

Example: cluster search (1)

- Given a number points in a 2 dimensional space, try and group spatially close points.
- Clustering is best if
  - Size of each cluster is minimal

Example: cluster search (2)

- Given a number points in a 2 dimensional space, try and group spatially close points.
- Clustering is best if
  - Size of each cluster is minimal



Example: cluster search (5)

```
void clusterize(point2d P[], int start, int end)
{
   // make a cluster entry for pair of points
   for each point X in P
         for each point Y in P
            if X != Y
                clusters += new cluster(X,Y)
   // see if the total surface area decreases when merging
   // with another area
   for each cluster A in clusters
         for each cluster B clusters
                  if A != B and surface(A)+surface(B) > surface(A+B)
                           clusters -= A
                           clusters += cluster(A+B)
}
```

Example: cluster search (6)

- Problems
  - Not recursive = not divide and conquer

Example: cluster search (7)

```
for each i in a do
        statement(i)
```

```
int len = length(a);
foo(a, len);
-----------------------------------------
void foo(Data []a, int len) {
        if (len == 1) statement(i)
        else {
                foo(a[0..len/2], len/2);
                foo(a[len/2 .. len-1], len/2);
        }
}
```

Questions

- Can each loop be rewritten to use recursion ?
  - ? YES: each jump to start of the loop = recursive invocation
- Can each loop be rewritten as above ?
  - ? NO: depends on data-dependencies

CILK (1)

- ANSI C extension
  - spawn call(parameters)
  - sync;
- For shared memory multiprocessors
- Translates CILK to plain C

```
cilk long parallel_fib(long n) {
        if(n < 2) return n;

        long x = spawn fib(n-1);
        long y = spawn fib(n-2);
        sync;
        return x + y;
}
```

CILK (2)

- Each CPU has a private 'job' stack
- Each spawn puts a descriptor on his job stack
  - The parameters of the spawned invocation
  - Where to store return value of call afterwards
- Each sync pops descriptors from his stack until stack exhausted
  - Because of divide & conquer
    - Big tasks on bottom of stack
    - Smallest tasks on top
  - Current processor picks work from top-of-the-stack
  - Others try and "steal" jobs from bottom of stack
    - "work stealing" algorithm

CILK (3)

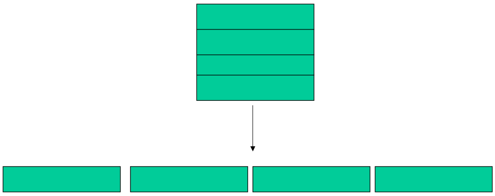
- Who do you steal jobs from ?
  - Random work stealing ?
  - Hierarchical work stealing ?
  - Match stealing pattern to that of network ?
- Do you sometimes push jobs to other machines when you have too many jobs ?
  - Push jobs to whom ?

CILK (4)

- Matrix multiplication using divide and conquer

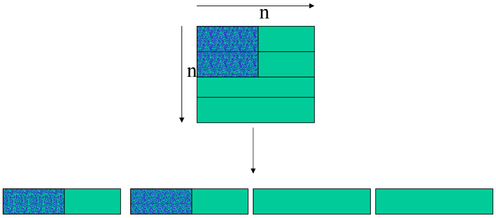
$$c_{ik} = \sum_{j=1}^{n} a_{ij} \times b_{jk}$$

CILK (4aa)
Inproving Matrix Memory Layout



Access element A[I][j] of NxN matrix using:
ptr + (N*I) + J

CILK (4ab)
Inproving Matrix Memory Layout



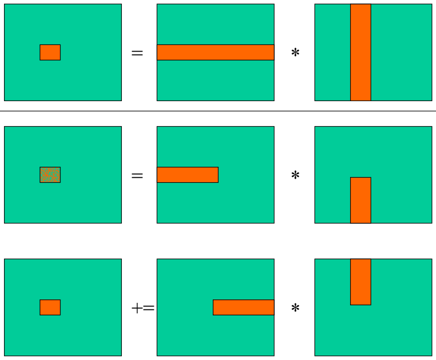
Access element A[I][j] of NxN matrix using:
ptr + (n*I) + J

CILK (4b)

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \times b_{jk}$$

```
void iter_matmul(double *A, double *B, double *C, int n)
{
    int i, j, k;

    for (i = 0; i < n; i++)
        for (k = 0; k < n; k++) {
            double c = 0.0;
            for (j = 0; j < n; j++)
                c += A[i * n + j] * B[j * n + k];
            C[i * n + k] = c;
        }
}
```

CILK (4c)



Is this the same

As this ?

CILK (4d)

```
cilk void rec_matmul(double *A, double *B, double *C,
        int m, int n, int p, int ld, boolean add)
{
    if ((m + n + p) <= 64) { /* base case */
        if (add) {
            for (int i = 0; i < m; i++)
                for (int k = 0; k < p; k++) {
                    double c = 0.0;
                    for (int j = 0; j < n; j++)
                        c += A[i * ld + j] * B[j * ld + k];
                    C[i * ld + k] += c;
                }
        } else {
            for (int i = 0; i < m; i++)
                for (int k = 0; k < p; k++) {
                    double c = 0.0;
                    for (int j = 0; j < n; j++)
                        c += A[i * ld + j] * B[j * ld + k];
                    C[i * ld + k] = c;
                }
        }
    } else spawn_sub_matrix_computations(A,B,C,m,n,p,ld,add);
}
```

CILK (4e)

```
void spawn_sub_matrix_computations(double *A, double *B, double *C,
                      int m, int n, int p, int ld, boolean add) {
     if (m >= n && n >= p) {
          int m1 = m / 2;
          spawn rec_matmul(A, B, C, m1, n, p, ld, add);
          spawn rec_matmul(A + m1 * ld,
                           B,
                           C + m1 * ld, m - m1,
                           n, p, ld, add);
     } else if (n >= m && n >= p) {
          int n1 = n  / 2;
          spawn rec_matmul(A, B, C, m, n1, p, ld, add);
          sync;
          spawn rec_matmul(A + n1,
                           B + n1 * ld,
                           C, m, n - n1, p, ld, 1);
     } else {
          int p1 = p / 2;
          spawn rec_matmul(A, B, C, m, n, p1, ld, add);
          spawn rec_matmul(A,
                           B + p1,
                           C + p1, m, n, p - p1, ld, add);
     }
}
```

CILK (6)

- Abort mechanism
  - During search space exploration you may want to abort all branches that have become superfluous
    - Dynamic programming does not stop already running superflous tasks, only prevents new unneeded tasks
  - One task kills
    - A sibling task
    - …and all its children
    - Where are those tasks running (other machines….)

JSR-166 (1)

- Future Java extension: ***java.util.concurrent.****
  - Instead of extending ***java.lang.Thread*** extend ***java.util.concurrent.FJTask***
  - FJTask instances are managed by ***java.util.concurrent.FJTaskRunner*** instances (themselves ***java.lang.Thread*** instances)

JSR-166 (2)

```
Step1: in class heading:
       class Fib extends FJTask {
       int number; // result & FIB number this task is to compute

Step2: in main:
       FJTaskRunnerGroup g = new FJTaskRunnerGroup(cpus);
       Fib f = new Fib(num);
       g.invoke(f);

Step 3: in run:
       int n = number;
       if (n <= 1) {
              // Do nothing: fib(0) = 0; fib(1) = 1
       } else if (n <= sequentialThreshold) {
             number = seqFib(n);
       } else {
              Fib f1 = new Fib(n - 1);
              Fib f2 = new Fib(n - 2);
              fork(f1); fork(f2);
              join();

              // Combine results:
              number = f1.number + f2.number;
       }
```

JSR-166 (3)

```
//Conceptually:


FJTask.fork(FJTask t)
        stack.push(t);

FJTask.join();
        while (! stack.empty())
                FJTask t = stack.pop();
                t.run();
```

Satin (1)

- An alternative Java extension for divide and conquer programming
- Designed for
  - Clusters of workstations
  - Clusters of clusters (aka the "computational grid")
- Objects implementing a 'Spawnable' interface have their methods invoked in parallel on other machines
- All parameters to methods of 'Spawnable' are call by copy
  - Distributed memory machines !

Satin (2)

```
interface FibInter extends satin.Spawnable {
    int fib(int n);
}

class Fib extends satin.SatinObject
                implements FibInter {
     public int fib(int n) {
                  if (n<2) return n;
                  long x = fib(n-1); // spawns new task
                  long y = fib(n-2); // spawns new task
                  sync();
                  return x + y;
}

public static void main(String args[]) {
                  Fib f = new Fib();
                  int result = f.fib(17);
                  f.sync();
                  System.out.println("result = " + result);
}
```

Satin (3)

- Implements
  - – Random stealing
    - Once a machine is idle
      - – try and perform "work stealing" from some other machine
  - – Cluster-aware random Stealing
    - Once a machine is idle
      - – Send a work request asychronously to a remote cluster
      - – Try and perform "work stealing from some other machine within own cluster in parallel to work request