

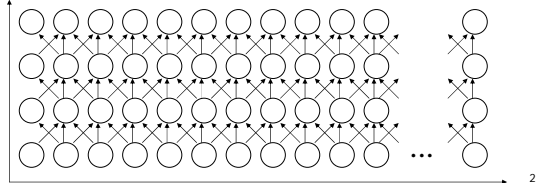
Static Clustering and Scheduling

Notions of Clustering, Scheduling, Granularity
 Clustering, Scheduling for Delay Model
 Clustering, Scheduling for LogP Model
 Malleable Task(-Graph) Scheduling

Example

```

1. for (t=1; t<T; t++) {
2.   forall (k=1; i<D; k++) in parallel {
3.      $\delta^{t+1}[k] = f(\delta^t[k-1], \delta^t[k], \delta^t[k+1])$ 
4.   }
5. }
    
```



Mapping to distributed systems

- Two principle approaches:
 1. Distribute the data
 - Each array element $\delta[k]$ is assigned to a processor
 - Computation follows
 2. Cluster/Schedule the computations
 - Each computation is scheduled individually
 - Consumed array elements $\delta[k-1]$ $\delta[k]$ $\delta[k+1]$ and produced array elements $\delta^{t+1}[k]$ become local variables of the tasks
 - Minimize overall execution time for a cost model
 - Clustering: for arbitrary many processors
 - Scheduling: for P processors

3

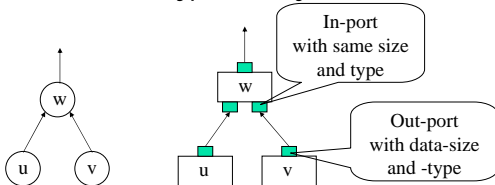
Definition: Task Graph

- A Task Graph $G=(V, E, \tau, c)$ is a directed, acyclic graph with
 - Nodes V representing computations, i.e. tasks
 - Edges E representing data-dependencies between tasks
 - Computation times $\tau: V \rightarrow \mathbb{N}$ representing the time for computing a task v
 - Communication size $c: E \rightarrow \mathbb{N}$ representing the time depending on the size in byte of data communicated via an edge e

4

Task Graph Implementation

- Task graphs are implemented a bit more complex to map to arguments and results of computation functions
- Ports with data types – only size needed



5

Problem

- Given a task Graph
 - Derived from a data parallel program
 - Directly programmed
- Different concrete target machines – modeled by cost model
 - Number of processors
 - Performance of processors
 - Net performance (latency, bandwidth)
- Minimize execution time automatically
 - **Clustering**: regardless of restricted number of processors
 - **Scheduling**: regarding restricted number of processors

6

Definition Clustering

- Given a task graph $TG=(N,E, \tau, c)$
- A clustering C is a assigns starting times $t(v)$ and processor numbers $p(v)$ to tasks v
- It is a relation $N \rightarrow (N, N)$ where
 - C is complete in N (each node gets at least one starting time $t(v)$ and processor number $p(v)$)
 - Starting times obey precedence relation and communication delay:
 - $(u,v) \in E \wedge (t, p) \in C(v) \Rightarrow$
 - $\exists (t', p') \in C(u): t' + \tau(u) + c(u,v) \leq t \vee$
 - $\exists (t', p) \in C(u): t' + \tau(u) \leq t$
- Completion time or Makespan:
 - $T(C) = \max \{t + \tau(v): (t, p) \in C(v) \text{ for any } v\}$

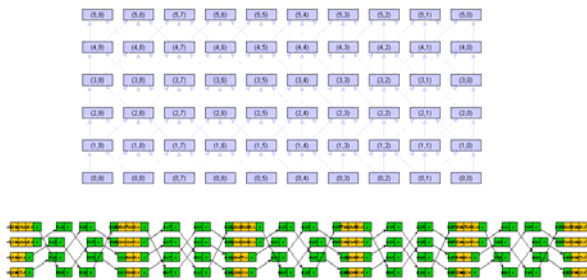
7

Definition Scheduling

- Given a task graph $TG=(N,E, \tau, c)$ and cost functions representing maximum costs c and L
- A schedule S is a relation $N \rightarrow (N, P)$ where
 - S is complete in N (each node gets a starting time $t(v)$ and a processor number $p(v) \in [1 \dots P]$)
 - $(u,v) \in E \wedge (t, p) \in S(v) \Rightarrow$
 - $\exists (t', p') \in S(u): t' + \tau(u) + c(u,v) \leq t \vee$
 - $\exists (t', p) \in S(u): t' + \tau(u) \leq t$
 (starting times obey precedence relation and communication delay)
- Completion time:
 - $T(S) = \max \{t + \tau(v): (t, p) \in S(v) \text{ for any } v\}$

8

Example Task Graph and Schedule



Optimization

- An **optimum clustering** C_{OPT} of a task graph TG is a clustering with
 - $T(C_{OPT}) \leq T(C)$ for all other clusterings C of TG
- An **optimum schedule** S_{OPT} of a task graph TG and a number of processors P is a schedule with
 - $T(S_{OPT}) \leq T(S)$ for all other schedules S of TG and P
- It is **NP hard** to find the optimum clustering and optimum schedule, resp., for a general task graph TG (even for the simplest cost models, e.g. uniform computation costs, no communication costs)

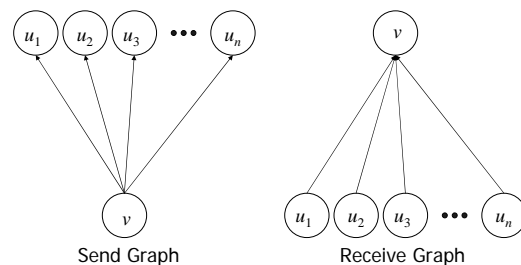
10

Optimization NP hard

- In practice ok?
 - Sometimes exponential solutions are acceptable if handy for practical relevant cases (optimum array distribution configuration – ILP solver)
 - Unfortunately not for clustering and scheduling
- Find special cases allowing optimum solutions
- Find approximation schemata
- Find approximations with constant factor approximations
- Heuristics are ultima ratio!

11

Send/Receive Task Graphs



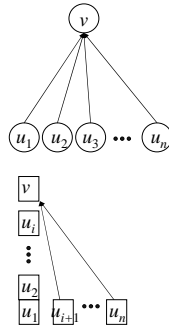
12

Cluster Receive Task Graphs

- Order all u_i non-increasingly in $\tau(u_i) + c(u_i, v)$
- Let $p(u_1) = p(v)$
- Stepwise add the first remaining task u_i to $p(v)$ until:

$$\sum_{j \in [1 \dots i]} \tau(u_j) \geq \tau(u_{i+1}) + c(u_{i+1}, v)$$

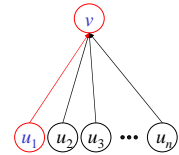
- Optimum solution!



13

Observation

- If $\min \tau(u_i) \geq \max c(u_i, v)$ only $p(u_1) = p(v)$
- All other tasks computed in parallel
- In general: parallel computation always pays
- Property of a task graph and cost functions under a certain cost model called **granularity**



14

Definition Granularity

- Ratio computation / communication costs
- In order to enable quantitative conclusions, we define:
 - Granularity $G(v)$ of a task v :

$$G(v) = \min(G_{in}(v), G_{out}(v))$$

$$G_{in}(v) = \min_{u_i \in Pred(v)} \tau(u_i) / \max_{u_i \in Pred(v)} c(u_i, v)$$

$$G_{out}(v) = \min_{u_i \in Succ(v)} \tau(u_i) / \max_{u_i \in Succ(v)} c(u_i, v)$$
 - Granularity $G(TG)$ of a task graph $TG = (N, E, \tau, c)$:

$$G(TG) = \min_{v \in N} G(v)$$
- Task graph TG is **coarse** grained iff $G(TG) \geq 1$
- Task graph TG is **fine** grained iff $G(TG) < 1$

15

Coarse Granularity Clustering

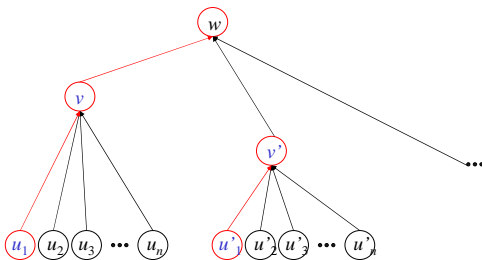
- For any coarse grained task graph TG a clustering $C(TG)$ with constant performance delay:

$$T(C) \leq 2 T(C_{OPT})$$
 can be found in polynomial time
- Constructive: naive clustering
 - layer-wise
 - Assign each task v to a separate processor at starting time $t(v) = \max_{u_i \in Pred(v)} t(u_i) + \tau(u_i) + c(u_i, v)$
- Proof (sketch):
 - $T(C_{OPT}) \geq \sum_{v \in \text{Longest Path in } TG} \tau(v)$
 - As each $\tau(v) \geq \max_{u_i \in Pred(v)} c(u_i, v)$ communications are smaller than $\sum_{v \in \text{Longest Path in } TG} \tau(v)$

16

Optimum Clustering Coarse Trees

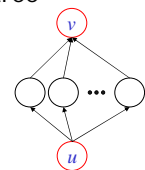
- Layer-wise, apply algorithm for receive graphs from the leaves to the root



17

Generalize on all task graphs

- For each output node of a TG generate tree of predecessors
- Introduces a lot of redundancy
- Find optimum clustering of each tree individually
- Unfortunately exponential if TG contains sub-structures like
- Trees have exponential many nodes



18

Example: Work opt. prefix sums

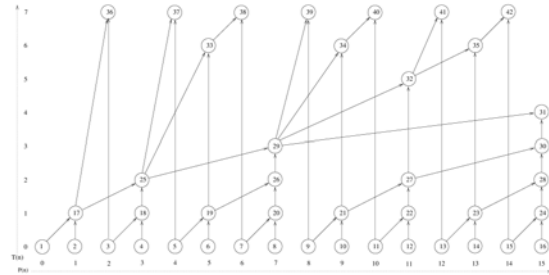
```

1. right[0..n]=[0..n];
2. for (i=1; i<n; i*=2) {
3.     forall (p=0; p<n; p++) in parallel {
4.         if ((p+1) mod i == 0)
5.             right[p]=right[p-i/2]+right[p];
6.     }
7. }
8. for (i=n; i<0; i/=2) {
9.     forall (p=0; p<n; p++) in parallel {
10.        if ((p+1) mod i == (i/2) & p>i)
11.            right[p]=right[p-i/2]+right[p];
12.    }
13. }

```

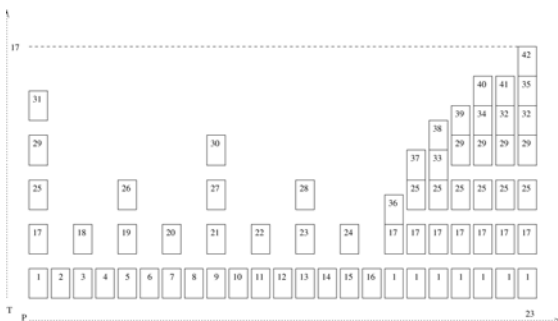
19

Example: Task graph prefix sums



20

Clustering depicted as Gantt-Chart



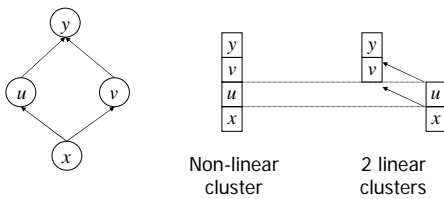
Linear Clustering

- Computing two tasks in direct precedence relation on the same processor cannot increase the overall computation time
- Any clustering constructed by merging the tasks of two processors only if all tasks are on a path in TG is called **linear**
- Any linear clustering C of a **coarse** grained TG guarantees $T(C) \leq 2 T(C_{OPT})$
 - Proof by showing that the linear clustering does not increase the completion time of the naive clustering
- If TG is **coarse** grained the optimum clustering is a linear one

22

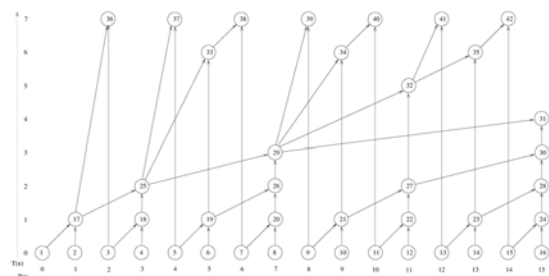
Optimum is Linear Clustering

Proof by iteration over
stepwise separation of independent tasks on a processor



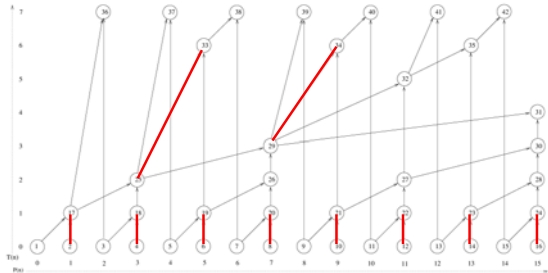
23

Linear clustering I



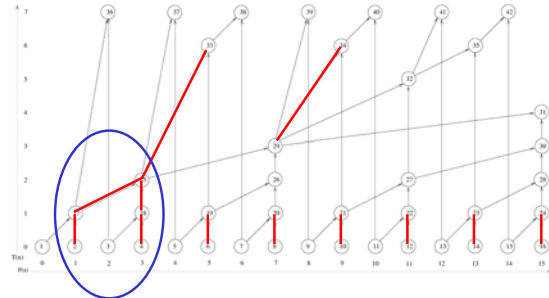
24

Linear clustering II



25

Non-Linear clustering



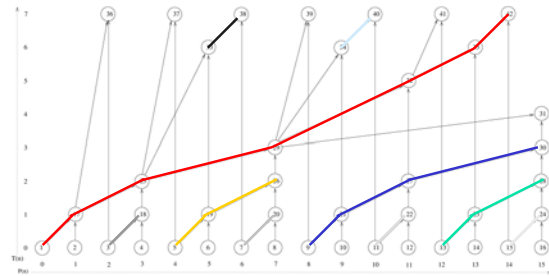
26

Heuristic: cluster longest path

- Computation dominated by longest (most expensive) path of task graph TG
- Idea: save communications on the longest (most expensive) path by computing its task on the same processor
- Iteratively remove the longest path from TG until its empty
- Factor 2 performance approximation and, as a heuristic, it usually saves time and processors (both not guaranteed)

27

Longest Path Linear clustering



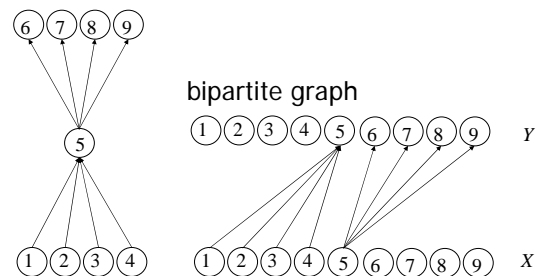
28

Min. Processor Linear Clustering

- Linear clustering is a node disjoint path cover Π
- Obviously $|\Pi| = P$
- Minimal path cover of $TG=(N,E)$ is maximum matching in bipartite graph:
 $M = (\{x_v | v \in N\} \cup \{y_v | v \in N\}, \{(x_u, y_v) | (u,v) \in E\})$
- Computable in polynomial time $O(|N|^{1/2} + |E|)$

29

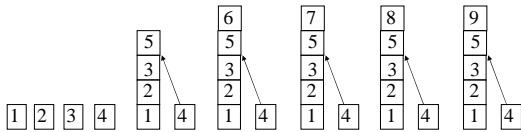
Example



30

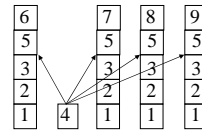
Example Schedule: $c=2, \tau=1$

Contains a lot of redundant computations



37

Obvious simplifications



- Only schedule output and sending tasks
- Compute each task sending its results only once.

38

Proof obligations

- $t_{\min}(v)$ is a lower bound for the optimum:
Show that **no task** can be scheduled before $t_{\min}(v)$
- $T(C) \leq 2 T(C_{\text{OPT}})$ guaranteed:
Show that **each task** can be scheduled before $2 t_{\min}(v)$

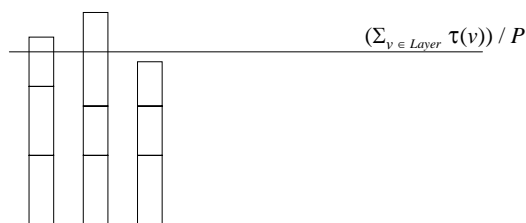
39

Brent Schedule

- Reduce the number of processors by greedy packing
- Layer-wise schedule computations
 - Let $t(p)$ be the completion time of computations on processor p (initially $t(p)=0$ for all processors)
 - Schedule task v on a processor p until $t(p)$ value larger $(\sum_{v \in \text{Layer}} \tau(v)) / P$
- Layer-wise schedule communications (if necessary)

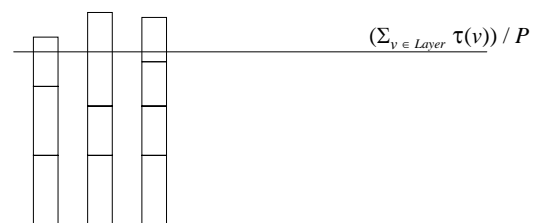
40

Schedule a layer



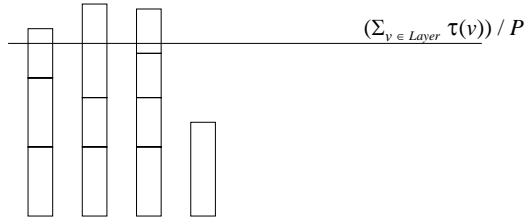
41

Schedule a layer



42

Schedule a layer



43

Performance of Brent Schedule

- Obviously requires maximal P processors as each processor does work $(\sum_{v \in Layer} \tau(v)) / P$ or more
- Error $\leq \max_{v \in Layer} \tau(v)$
- Schedule of each layer $T(S) \leq 2 T(S_{OPT})$ since:
 - $T(S_{OPT}) \geq \sum_{v \in \text{Longest Path in } TG} \tau(v) = \max_{v \in Layer} \tau(v)$
 - $T(S_{OPT}) \geq (\sum_{v \in TG} \tau(v)) / P = (\sum_{v \in Layer} \tau(v)) / P$
- Ignoring communication costs and assuming uniform computation costs in the layers: $T(S) \leq 2 T(S_{OPT})$
- Communication cost additionally $1/G \max_{v \in Layer} \tau(v)$
- With uniform communication cost: $T(S) \leq (2+1/G(TG)) T(S_{OPT})$

44

Generalization for LogP

- Somewhat harder as communication costs processor time
- Makes it impossible to simply generalize results on delay model
- Correct, but too conservative to set:
 - Computation $\tau_{LogP} = \tau + 2o + (odg(v) + idg(v) - 2) \max(g, o)$
 - Communication $\tau_{LogP}(u, v) = L$

45

Send/Receive graphs (revisited)

- Optimum polynomial time solutions for delay model
- NP hard* to find the optimum *LogP* schedule
- There is an $|N|^2$ algorithm computing a receive graph schedule with performance guarantee $T(S_{LogP}) \leq (10/3 - 1/(3P)) T(S_{LogP-OPT})$
- There is an $|N|^2 \max(\log |N|, P^2)$ algorithm computing a send graph schedule with performance guarantee $T(S_{LogP}) \leq (7/3 - 1/(3P)) T(S_{LogP-OPT})$

46

Receive graph schedule (sketch)

- Compute a schedule allowing exactly k receive operations for $k = 0 \dots n$ and choose then the minimum one:
 - Order all u_i non-increasingly in $\tau(u_i)$
 - Schedule tasks on P processors by longest processing time minimizing heuristic (greedy) until k tasks are scheduled on $P-1$ processors or all tasks are scheduled
 - Schedule remaining (if any) on last processor
 - Schedule v on last processor
 - Schedule communications to last processor from first k tasks (scheduled to the first $P-1$ processors)

47

Performance (proof sketch)

- Join of k items, must be received sequentially and compute the final task $2o + L + (k-1) \max(g, o) + \tau(v) \leq T(S_{OPT}(k))$
- Schedule first k tasks on $P-1$ processors: $T(k) = (4/3 - 1/(3P)) T(S_{OPT}(k))$
 $T(S(k)) \leq T(k) + T(S_{OPT}(k))$
- Schedule last $n-k$ tasks on one processor sequential and, hence, optimally: $T(S(k)) = (4/3 - 1/(3P)) T(S_{OPT}(k)) + 2 T(S_{OPT}(k))$
- Overall schedule selects minimum $T(S(k))$: $T(S) = (10/3 - 1/(3P)) T(S_{OPT}(k))$

48

Generalization of Granularity

- Generalize from delay model
 $L(u, v) = 2o + L + (odg(u) + idg(v) - 2) \max(g, o)$
- Granularity $G(v)$ of a task v :
 $G(v) = \min(G_{in}(v), G_{out}(v))$
 $G_{in}(v) = \min_{u_i \in Pred(v)}: \tau(u_i) / \max_{u_i \in Pred(v)} L(u_i, v)$
 $G_{out}(v) = \min_{u_i \in Succ(v)}: \tau(u_i) / \max_{u_i \in Succ(v)} L(u_i, v)$
- Granularity $G(TG)$ of a task graph $TG = (N, E)$:
 $G(TG) = \min_{v \in N}: G(v)$
- Task graph TG is **coarse** grained iff $G(TG) \geq 1$
- Task graph TG is **fine** grained iff $G(TG) < 1$

49

Coarse Granularity

Computations dominate:

- Brent schedule computations
- All to all communication between two layers
- Sparse out unnecessary communication

Performance:

$$4 T_{OPT}(G) \geq T_{LogP}$$

50

Fine Granularity

Communication dominates

- Delay communication until computation dominates
 - Allow imbalanced computation
 - Allow redundant computation
- Bundle communication
 - LogP model with communication functions
 - $L_{max}(n) < n L_{max}(1)$
- Within the remaining freedom
 - Balance computations
 - Eliminate unnecessary redundant computation

51

Results

- Balanced Trees
 $(3 + \epsilon) T(S_{OPT}) \geq T(S)$
- Balanced task graphs
 $(4 + \epsilon) T(S_{OPT}) \geq T(S)$
- Affine Index computations
 $(4 + \epsilon) T(S_{OPT}) \geq T(S)$

52

Conclusion

- Only approximations of the optimum
 - Compilers can give good results in practice, sometimes not good enough
 - Therefore programmers must be able to find better solutions for specific problems "by hand"
- Problem:
 - Scheduling of several applications, i.e., several schedules
 - Large task graphs from program sample execution lead to large optimization times
- Solution:
 - malleable task(-graph) scheduling

53

Problem I – Global Scheduling

- Given a global scheduling problem defined by a machine LogP model instance $L(x), o(x), g(x), P$ and number of (local) LogP schedules S_1, \dots, S_n corresponding to task graphs TG_1, \dots, TG_n
- Each local schedule S_i is for a fixed number $p_i \in \{1, \dots, P\}$ processors and optimized regardless of the others
- Couldn't we regard the other scheduling problems when fixing p_i ?

54

Trivial Strategies

- **Strategy 1:** the schedule S_1, \dots, S_n for each task graphs TG_1, \dots, TG_n must only use one processor
 - Global scheduling reduces to local scheduling of n independent tasks
 - Obviously suboptimal, e.g., when $n \bmod P \neq 0$
- **Strategy 2:** the schedule S_1, \dots, S_n for each task graphs TG_1, \dots, TG_n must use P processors
 - Then sequentially execute the schedules
 - Obviously suboptimal, e.g., when a TG_i can only exploit $p_i < P$ processors optimally (at all or in some layers, especially the first and last)
- Both extreme strategies are suboptimal

55

Idea

- **Malleable Tasks**
 - Task that can be executed by more than one processor
 - Weight function: $T : \{1, \dots, P\} \rightarrow \mathbb{N}$
 - Assumption: T is non-increasing and has no super-linear speed-ups
- For each task graph TG_i , P local schedules $S_i(p)$ are computed, one for each $p \in \{1, \dots, P\}$
- Weight function: $T(p) = T(S_i(p))$
- Global scheduling corresponds to scheduling independent malleable tasks

56

Results

- Problem is NP-hard
- Mounie, Rapine, Trystram; 1999:
 - n independent malleable tasks
 - algorithm computes in time $O(n \cdot P)$ a schedule with performance guarantee $\sqrt{3}$, if $P \leq n^2$
- Decker, Lücking, Monien; 2003:
 - n identical independent malleable tasks can be optimally scheduled on P processors in time $O((n \cdot 2T(1))^P)$ where $T(1)$ is the execution time of the task on one processor.
 - Furthermore, there is a $5/4$ -approximation (performance guarantee) algorithm with execution time $O(P^3 + \log n)$

57

Problem II –Scheduling Efficiency

- A task graph (by definition) is
 - Non-hierarchical, i.e., no refinements of tasks to sub-task graphs, no procedures, no library code
 - Acyclic, i.e. no loops
- Explosion of program size when computing a task graph from a program:
 - Complete procedure inlining
 - Complete loop unrolling

58

Idea

- Procedures are task graphs that can be executed on more than one processor, i.e., they are **Malleable Task Graphs**
- Tasks are
 - Atomic computations (tasks)
 - Procedure calls are calling other malleable task graphs
- Schedule in the reverse order of the call graph – starting with leaf procedures (which don't contain any further calls)
- Schedule layer-by-layer using independent malleable task scheduling
- Inlining can be avoided - library routines can be integrated naturally

59

Parallel Programs - Malleable Tasks

- Each procedure including main corresponds to a malleable task
- Data-dependencies define for each procedure a task-graph
- Some operation/tasks are procedure calls referring to other procedures, i.e. to other malleable tasks
- Numeric operations, assignments etc. are atomic tasks.

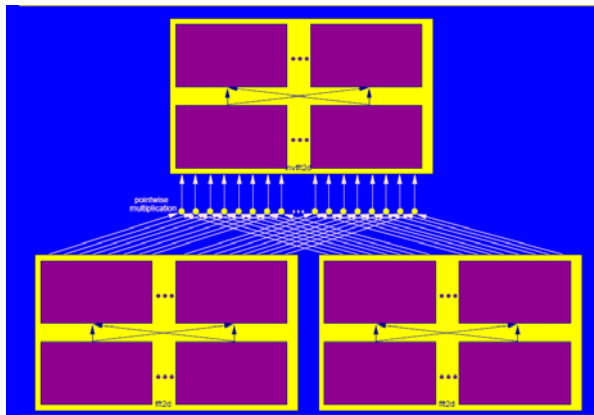
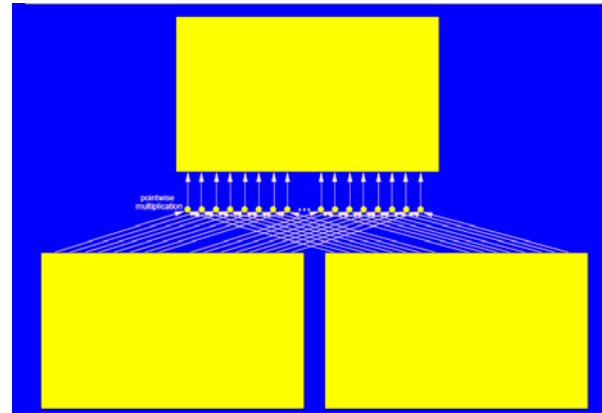
60

Example: Convolution

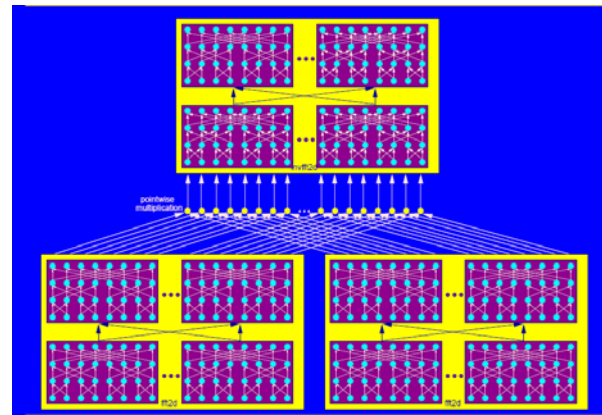
```

fun fft(v:array[n] of complex, ω: complex): array[n] of complex
//r(i) denotes the value of the reversed bit representation of i.
  for i=0...n-1 do in parallel v[i] :=v[r(i)]; end
  for j=0...log n-1 do
    for k:=0...n/2j-1 do in parallel
      for i:=0...2j-1 do in parallel
        pardo
          v[k*2j+1+i] := v[k*2j+i] + ωi*n/2j+1 v[(2k + 1)*2j+i];
          v[(2k + 1)*2j+i] := v[k*2j+i] - ωi*n/2j+1 v[(2k + 1)*2j+i]
        end pardo
      end
    end
  end
fun fft2d(a:array[n,n] of complex, ω:complex): array[n,n] of complex
  for i=0...n-1 do in parallel a[* ,i] := fft(a[* ,i], ω); end
  for i=0...n-1 do in parallel a[i,*] := fft(a[i,*], ω); end
  return a;
fun convolution(a,b:array[n,n] of complex, ω:complex): array[n,n] of
  complex
  pardo a := fft2d(a, ω); b:= fft2d(b, ω); end
  for i,j=0...n-1 do in parallel c[i,j] := a[i,j] * b[i,j]; end
  return invfft2d(c, ω);
  
```

61



62



63

Hierarchical Task Graph

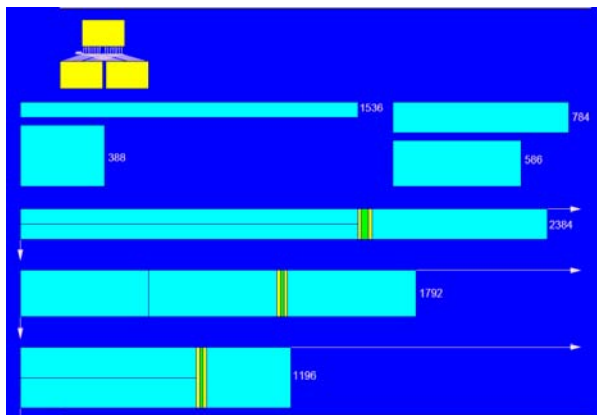
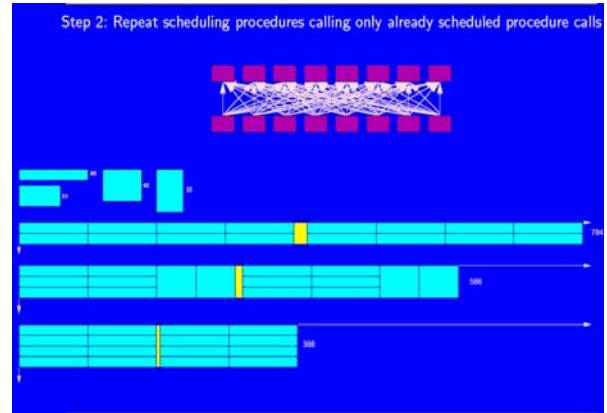
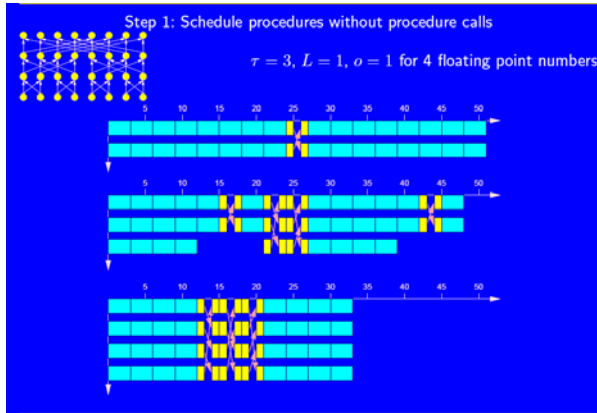
- Atomic Task v : Task that can be executed only by one processor, i.e., τ_v is a constant
- Task Graph: containing only atomic tasks
- Hierarchical Malleable Task Graphs: (HMT-Graph) Inductively defined as follows:
 - Any task graph $TG=(V, E, \tau, c)$ with only atomic tasks is an HMT-graph
 - A directed acyclic graph $G=(V, E, T, c)$ is an HMT-graph iff each $v \in V$ is an HMT-graph and $T = \{\tau_v : \{1, \dots, P\} \rightarrow \mathbb{N} \mid v \in V\}$
 - $v \in V$ is atomic or itself a HMT-graph $G_v=(V_v, E_v, T_v, c_v)$
- $\tau_v(1) = \sum_{x \in V_v} \tau_x(1)$

65

Scheduling HTGs to P processors

- Each procedure is scheduled in topological order of the hierarchy defined by the call graph, beginning with the atomic tasks (corresponding leaf procedures that do not invoke others)
- Base-Case: Procedures without procedure call tasks can be scheduled using an arbitrary algorithm for tasks graph scheduling (scheduling as before)
 - Schedule for $p = 1, \dots, P$ processors
 - Note, that this yields the function $\tau(p)$ for tasks calling this procedure
- Inductive Case: Procedure with procedure call tasks
 - Each procedure call v corresponds to a malleable task $G_v=(V_v, E_v, T_v, c_v)$
 - Schedule for $p = 1, \dots, P$ processors and layer by layer apply a scheduling algorithm for scheduling independent malleable tasks,
 - Schedule the communications between the layers

66



Results

- Scheduling of malleable tasks graphs
- Assume the scheduling algorithm for the base case has a performance guarantee of c (c approximation)
- Furthermore, let k be the hierarchy depth and the μ degree of malleability of the malleable task graph (intuitively: minimum of work in a layer divided by work in that layer + costs for subsequent communications)
 - $T(S(G)) \leq c \cdot 3^{k/2} / \mu T(S_{OPT}(G))$
- Papers published on the course home page

Conclusion

- Problem I:
 - Malleable task scheduling of independent tasks leads to good global scheduling results
- Problem II:
 - Classic local scheduling leads to good speed-ups but is difficult to integrate into compilers
 - Avoidance of procedure inlining: malleable task scheduling
 - Malleable task scheduling weakens performance guarantee but reduces the complexity of local scheduling
 - Good for "large" (many call sites but only few procedures) malleable task graphs

71

70