Parallelizing with xDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems

Dounia KHALDI

CRI, Mathématiques et systèmes
MINES ParisTech

June 19, 2012

Problem Statements

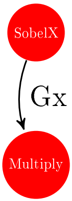
Evolution of the architecture (Multicores, GPUs...)

Evolution of parallel execution environments (OpenMP, MPI, OpenCL...)

Parallel software developed by converting sequential programs by hand

Context

```
in = InitHarris();
//Sobel
SobelX(Gx, in);
SobelY(Gy, in);
//Multiply
MultiplY(Ixx, Gx, Gx);
MultiplY(Iyy, Gy, Gy);
MultiplY(Ixy, Gx, Gy);
//Gauss
Gauss(Sxx, Ixx);
Gauss(Syy, Iyy);
Gauss(Sxy, Ixy);
//Coarsity
CoarsitY(out, Sxx, Syy, Sxy);
```



- Scheduling ⇒ minimize completion time.
- length(path) = communication_cost(edges)+computational_cost(nodes).
- Dynamic vs. Static.
- List-scheduling heuristics.

List-Scheduling Processes

- Priorities are computed of all unscheduled nodes:
  - Top level (tlevel(τ)): length of the longest path from the entry node to $\tau \Rightarrow$ earliest possible start-time.
  - Bottom level (blevel(τ)): length of the longest path from τ to the exit node $\Rightarrow$ latest start-time = CriticalPathLength - blevel(τ).
- The node τ with the highest priority is selected for scheduling.
- τ is allocated to the cluster that offers the earliest start-time.

task	tlevel	blevel
τ_4	0	7
τ_3	3	2
τ_1	0	5
τ_2	4	3

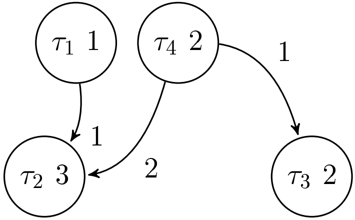


Figure: A Directed Acyclic Graph

DSC (Dominant Sequence Clustering)

[Yang and Gerasoulis 1994]

- priority(τ) = tlevel(τ) + blevel(τ).
- A zeroing(τ_p, τ) puts τ in the cluster of a predecessor τ_p $\Rightarrow$ reduces tlevel(τ) by setting to zero the cost of the incident edge (τ_p, τ).

step	task	tlevel	blevel	DS	scheduled tlevel	
					κ_0	κ_1
1	τ_4	0	7	7	0*	
2	τ_3	3	2	5	2*	3
3	τ_1	0	5	5		0*
4	τ_2	4	3	7	5	4*

κ_0	κ_1
τ_4	τ_1
τ_3	τ_2

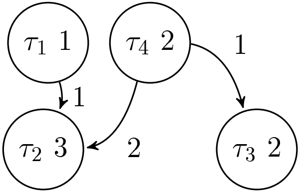


Figure: A Directed Acyclic Graph

DSC Algorithm Weaknesses

- Number of processors is not predefined → blind clustering.
- Memory size is not predefined → blind clustering.
- Creates a new cluster when no zeroing is accepted → creates long idle slots in already existing clusters.

⇒ xDSC: A MEMORY-CONSTRAINED, NUMBER OF PROCESSOR-BOUNDED EXTENSION OF DSC

xDSC: Resource Constraint Warranty

1. Memory Constraint Warranty (MCW):
   - Verifying that the zeroing does not exceed a memory threshold M.
   - task_data(τ) is an overapproximation of the amount of memory used by Task τ.
   - cluster_data(k) is an overapproximation of the amount of memory used by Cluster k.
   - size_data(data_merge (cluster_data (k) , task_data (τ))) $\leq$ M.
2. Bounded number of processors:
   - Verifying that new allocations do not exceed a cluster number threshold P.
   - cluster_time(k) is the start time of the last scheduled task in k + its task_time.
   - otherwise, $\text{argmin}_{\text{k}\in\text{clusters}}$ cluster_time(k) under the constraint MCW.

xDSC: Efficient Allocation

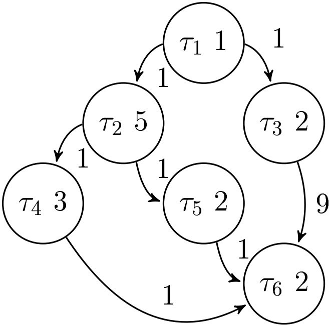


Figure: A DAG amenable to cluster minimization

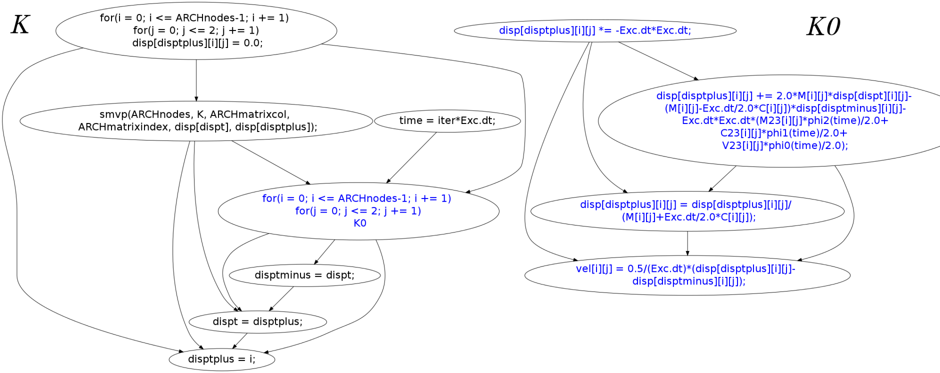
- Allocation of τ to the last idle slot of κ,
- Decreases tlevel(τ).
- For all nodes τ_s in κ:
  - scheduled(successors(τ_s)),
  - successors(τ) are included in successors(τ_s).

step	task	t level	b level	DS	tlevel κ_0	tlevel κ_1
1	τ_1	0	15	15	0*	
2	τ_3	2	13	15	1*	
3	τ_2	2	12	14	3	2*
4	τ_4	8	6	14		7*
5	τ_5	8	5	13	8*	10
6	τ_6	13	2	15	10*	

κ_0	κ_1	κ_2
τ_1		
τ_3	τ_2	
	τ_4	τ_5
τ_6		

κ_0	κ_1
τ_1	
τ_3	τ_2
τ_5	τ_4
τ_6	

DAG: Hierarchical Clustered Dependence Graph (KDG)



- A test (both branches: true+false) constitutes one node (task).
- A loop nest is an indivisible node.
- A simple instruction is an indivisible node.
  $\Rightarrow$ Hierarchy: recursively include KDGs.

1. Edge Cost:
   - Number of bytes of dependences RAW to annotate edges in the KDG,
   - edge_cost(τ_i, τ_j) = size_data(regions_intersection
     (read_regions(τ_i), write_regions(τ_j))).
2. Task Data:
   - task_data(τ) = data_merge(read_regions(τ), write_regions(τ))
   - data_merge(R_1,R_2) =
     regions_union(R_1,R_2) - regions_intersection(R_1,R_2)
3. Size of regions (convex polyhedra) $\Rightarrow$ Ehrhart polynomials represent the number of integer points contained in a given parameterized polyhedron.
4. Task Time:
   - An estimation of complexity for each node in the KDG,
   - task_time(τ) = complexity_estimation(τ) $\Rightarrow$ Polynomials.

Experiments

1. Applications
    - Signal processing application ABF (Adaptive Beam Forming) [Griffiths 1969].
    - Image processing application Harris corner detector [Harris and Stephens 1988]: detect the point of interest in an image.
    - SPEC benchmark equake [Bao et al. 1998]: simulation of seismic wave propagation in large valleys.
2. Machines
    - SMP: 2-socket AMD quadcore Opteron with 8 cores, M = 16Gb of RAM, 2.4 GHz.
    - DMP: 6 bicore processors Intel(R) Xeon(R), M = 32Gb of RAM per processor, 2.5 GHz.

- When data are known numerical parameters, then each task polynomial is a constant (case of the application ABF).
- However, when input data are unknown at compile time (case of the application Harris), we use a simple heuristic to check the behavior of that polynomials, by comparing the coefficients of their monomials.
- Assume that all polynomials are monomials on the same bases.

Function	**Complexity (polynomial)**	**Static time estimation**
InitHarris	9 × sizeN × sizeM	9
SobelX	60 × sizeN × sizeM	60
SobelY	60 × sizeN × sizeM	60
MultiplY	20 × sizeN × sizeM	20
Gauss	85 × sizeN × sizeM	85
CoarsitY	34 × sizeN × sizeM	34
One image transfer	4× sizeN × sizeM	4

From Polynomial to Values

Instrumentation

```
FILE *finstrumented = fopen("instrumented_equake.in","w");
...
fprintf(finstrumented, "62␣=␣%ld␣\n", 179 * ARCHelems + 3);
for (i = 0; i < ARCHelems; i++){
  for (j = 0; j < 4; j++)
    cor[j] = ARCHvertex[i][j];
  ...
}
...
fprintf(finstrumented, "163␣=␣%ld␣\n", 20 * ARCHnodes + 3);
for(i = 0; i <= ARCHnodes-1; i += 1)
  for(j = 0; j <= 2; j += 1)
    disp[disptplus][i][j] = 0.0;
fprintf(finstrumented, "163␣->␣166␣=␣%ld␣\n", ARCHnodes * 9); //edge_cost(163,166)
fprintf(finstrumented, "166␣=␣%ld␣\n", 110 * ARCHnodes + 106); //task_time(166)
smvp_opt(ARCHnodes, K, ARCHmatrixcol, ARCHmatrixindex, disp[dispt], disp[disptplus]);
fprintf(finstrumented, 167, 6);
time = iter*Exc.dt;
fprintf(instrumented_equake, 168, 510.50 * ARCHnodes + 3);
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++){
    disp[disptplus][i][j] *= -Exc.dt*Exc.dt;
    disp[disptplus][i][j] +=
      2.0*M[i][j]*disp[dispt][i][j]-M[i][j]-Exc.dt/2.0*C[i][j])*disp[disptminus][i][j] -
      Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
      C23[i][j] * phi1(time) / 2.0 + V23[i][j] * phi0(time) / 2.0);
    disp[disptplus][i][j] = disp[disptplus][i][j] / (M[i][j] + Exc.dt / 2.0 * C[i][j]);
    vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j] - disp[disptminus][i][j]);
  }
fprintf(finstrumented, "175␣=␣%d␣\n", 2);
disptminus = dispt;
fprintf(finstrumented, "176␣=␣%d␣\n", 2);
dispt = disptplus;
fprintf(finstrumented, "177␣=␣%d␣\n", 2);
disptplus = i;
```

Experiments: ABF and equake

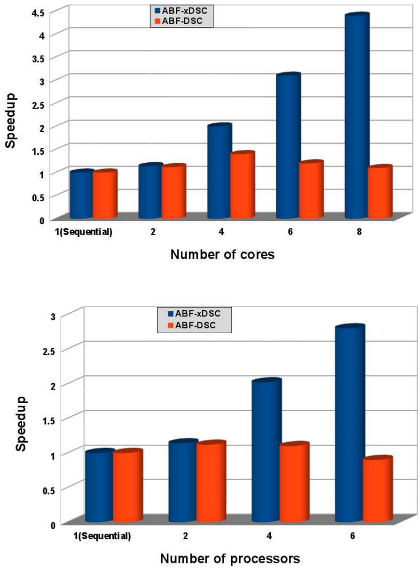


Figure: OpenMP/MPI vs. sequential speedup (ABF)

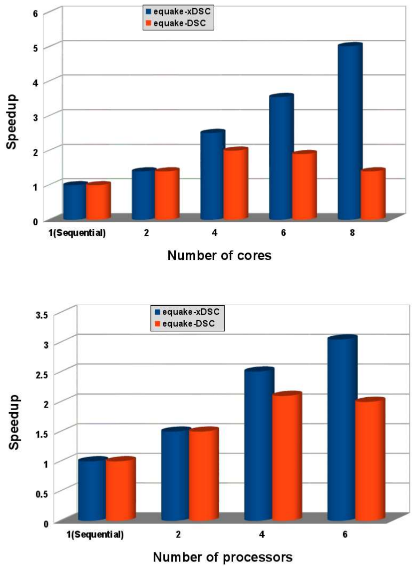
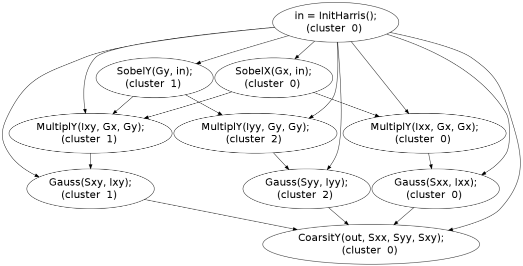
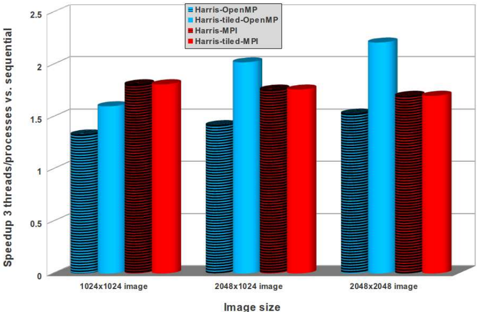


Figure: OpenMP/MPI vs. sequential speedup (equake)

Experiments: Harris





Conclusion

- xDSC: a new static scheduling,
- Precise and efficient cost model,
- Targeting both shared and distributed memory architectures,
- Memory constraint, Bounded number of processors, Efficient processor allocation.

Future Work

- Automatic code generation : OpenMP + MPI.
- Efficient hierarchical processor allocation strategy in order to yield a better xDSC-based parallelization process

Parallelizing with xDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems

Dounia KHALDI

CRI, Mathématiques et systèmes
MINES ParisTech

June 19, 2012