Granularity Control in Logic Programs

Saumya K. Debray
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

Granularity Control: Motivation

- Advantages of declarative programming languages:
  - Language model does not explicitly specify low-level operational semantics.
  - Implementations have freedom to choose execution strategy.
  - This makes it possible to exploit parallelism in a simple way.

    In principle, everything can be executed in parallel.
- In practice, just because something *can* be done in parallel does not mean it *should* be done in parallel.
- Overheads of parallel execution should be taken into account.

Granularity Control: Example

- Consider the clause

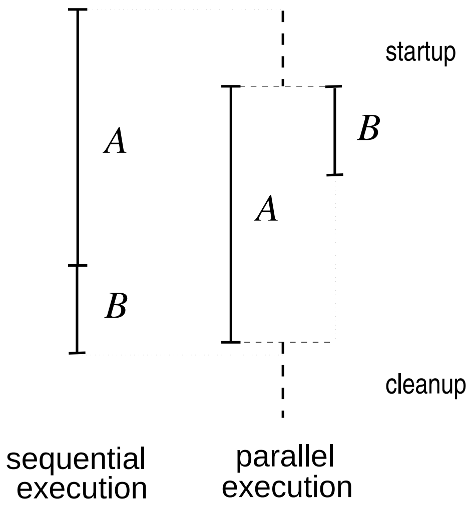
  `part([X|L],Y,[X|Sm],Lg) :- X < Y, part(L,Y,Sm,Lg).`
  - Assume args 1 and 2 are ground. Then the goals

    `X < Y` and `part(L,Y,Sm,Lg)`

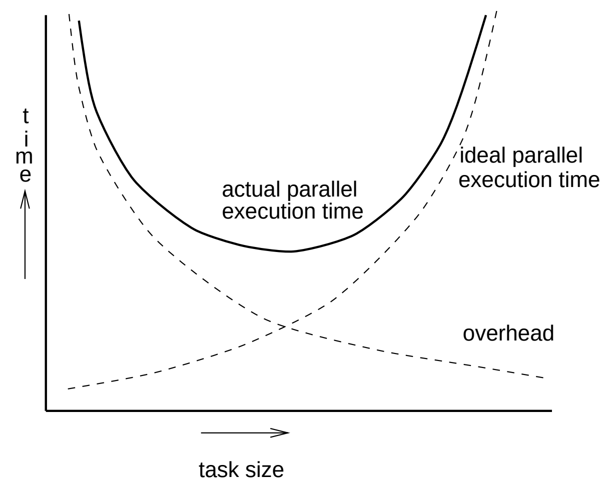
    are independent and can be executed in parallel.
  - The execution of `X < Y` takes 1 or 2 machine instructions.

    Spawning a separate thread for this goal is considerably more expensive.
  - It is cheaper to execute these goals sequentially rather than in parallel.
- In general: execute sequentially if the sequential execution cost is less than the parallel execution cost.

Granularity Control : cont'd



Parallelism vs. Overhead: The Tradeoff



Approaches to Granularity Control

Static Partitioning : partitioning decisions made entirely at compile time.

Examples : Kruatrachue & Lewis 88, Sarkar 89.

Advantage : Low runtime overhead.

Disadvantage : Lack of flexibility.

Dynamic Partitioning : Partitioning decisions made at runtime.

Examples : Tick 88, Debray *et al.* 90.

Advantage : Flexibility.

Disadvantage : Possible runtime overhead.

Adaptive Strategies : partitioning decisions made dynamically if resources permit this.

Examples : Lazy task creation [Mohr *et al.* 91].

Advantage : flexibility, low runtime overhead.

Disadvantage : no "cutoff" for tasks that are too small to justify parallel execution.

Static Partitioning: Kruatrachue 88

- Assumes static acyclic task graphs where all execution and communication costs are known.
- *Basic Approach* :
  1. Start with the finest granularity partitioning.
  2. Distribute tasks to processors (taking communication costs into account) so as to minimize overall execution time.

     Duplicate computations where appropriate to reduce communication costs.
  3. For each processor, pack the tasks mapped to that processor into larger sequential threads.
- Determination of computation and communication costs unaddressed.
- Suitable for static computations, e.g., numerical computations in Fortran.

  Not suited for highly dynamic computations, e.g., symbolic computations in Prolog.

Profile-Based Partitioning: Sarkar 89

- Static partitioning method:
  - Use execution profiles to estimate the computational cost of a procedure.
  - Use these cost estimates to guide parallelization decisions.
- Since partitioning decisions are static, they cannot be adapted to runtime needs, e.g.: input size or system load.

  Example : Consider the program

```
qsort([H|L],R) :-
  part(H,L,Sm,Lg),
  qsort(Sm,SmS),
  qsort(Lg,LgS),
  append(Sm,[H|LgS],R).
qsort([],[]).
```

The procedure `qsort` will be be executed either always in parallel, or always sequentially.

Sarkar's Partitioning Algorithm

Analytical Model : For a given program p,

$$T(p) = max \begin{cases} T_{crit} \\ (T_{seq} + \sum_S freq(S) \cdot overhd(S))/P \end{cases}$$

where

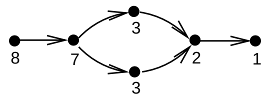
S is the set of parallel subtasks;
P = no. of processors.

Algorithm :

1. Start with the finest granularity partitioning. Represent this as a task graph.
2. Repeat until no further merging possible:
   (a) Pick the subgraph B with largest (weighted by execution frequency) overhead.
   (b) For each subgraph C of B, estimate the execution time resulting from the merging of B and C.
   (c) Choose the C that produces the smallest execution cost and merge it with B.
3. Use a history of the costs of different partitions obtained during these iterations to identify and reconstruct the cheapest one.

Dynamic Granularity Control: Tick 88

- Dynamic weight-driven scheduling.
- Associates a "weight" with each procedure based on its depth in the call graph.



These weights correspond to the nesting depth of procedures. No attempt is made to actually estimate runtime costs.

- Addresses *scheduling* rather than *partitioning* :
  - At runtime, when an idle processor requests a task from the scheduler, it gets one with maximal priority.

  This corresponds, roughly, to parallelizing "outer computations" (i.e., closer to the root of the computation tree) first.
- Parallel execution is not cut off even when parallelism is clearly not profitable.

Refinements: Tick *et al.* 1992

Goals :

- Cheap estimation of the relative costs of goals.
- Cost estimates for a procedure take an *iteration parameter* that captures the overall size of its inputs.
- These cost estimates provide *runtime priorities* that are used to guide scheduling:
  - At runtime, when an idle processor requests a task from the scheduler, it gets one with maximal priority.
- *Main Issue* : How to distribute the iteration parameter from the head of a clause to its body goals.
  - Relies on several heuristics.
  - Main idea: if there are k recursive body goals and the input is of size n, then each recursive call gets an input of size n/k.

Size-Based Dynamic Granularity Control

- Cost of a computation typically depends on the input size.

  ⇒ Partitioning decisions cannot always be made at compile time.
- Structure of code with dynamic granularity control:

  if (. . . *size test*. . . **) then**
      execute in parallel
  else
      execute sequentially

  Want to minimize runtime overhead associated with granularity control.

  ⇒ Should do as much of the work as possible at compile time.
- *Correctness considerations* : what invariants can we give about the compile-time analyses and optimizations?

Size Based Granularity Control: Debray *et al.* 90

- Based on estimating the runtime cost of a computation.
- Dynamic granularity control: the actual partitioning decisions are made at runtime.
- To minimize runtime overhead, as much of the work as possible is done at compile time.
  - At compile time, use cost analysis to determine the cost function for each procedure.
  - At runtime, test input sizes to decide parallel execution.
- Suppose the cost of a procedure p is $\mathcal{C}_p(n)$. Let the system overhead for task creation and management be K.

  A computation $p(\bar{x})$ in a program is transformed, conceptually, to

  if $\mathcal{C}_p(size(\bar{x})) > K$ **then**
  spawn a parallel thread for $p(\bar{x})$
  else
  execute $p(\bar{x})$ sequentially

Issues in Size Based Granularity Control

1. Runtime overhead of computing the cost function $\mathcal{C}_p(n)$.
2. Runtime overhead of computing data structure sizes.
3. Runtime overhead of granularity control tests.

Reducing Overheads 1: Cost Function Computation

- Suppose the cost function for a procedure p is

  $\mathcal{C}_p(n) = 3n^2$

  and the task management overhead is $K = 48$.
- Conceptually, code for a call $p(\bar{x})$ has the form

  $n = size(\bar{x})$;
  if $3n^2 > 48$ **then** ... parallel...
  else ... sequential...
- This can be simplified at compile time to

  $n = size(\bar{x})$;
  if $n > 4$ **then** ... parallel...
  else ... sequential...
- In general, we can compute a threshold K_p for each procedure p with cost function $\mathcal{C}_p$ as follows:

  $K_p = 0$;
  while $(\mathcal{C}_p(K_p) \leq K)$ **do**
  $K_p = K_p + 1$;
  od

Reducing Overheads 2: Size Computations

- Consider granularity control of quicksort:

  ```
  qsort([H|L],R) :-
      part(L,H,Sm,Lg),
      if (length(Sm) > 10) then ...parallel...
      else ...sequential...
  ```

  The input list is traversed repeatedly to compute *length*(L).
- *Basic Idea* [García & Hermenegildo ICLP 95] :
  - Compute data structure sizes incrementally as the structure is being constructed.
  - Propagate size information via source-level transformations.
- *Example* :

  ```
  part([X|L],Y,[X|Sm],Lg,SmSz,LgSz) :-
      X < Y,
      part(L,Y,Sm,Lg,SmSz1,LgSz),
      SmSz is SmSz1 + 1.

  qsort([H|L],R) :-
      part(L,H,Sm,Lg,SmSz,LgSz),
      if (SmSz > 10) then ...parallel...
      else ...sequential...
  ```

Dynamic Term Size Computation

- All programs in a program are either read in or created by the program.

  ⇒ Every term is traversed at some point before it is used.

- Given a term whose size is needed at a program point:
  - Find a procedure that traverses the term at an earlier point.
  - Transform it to also compute the term size "on the fly."
- This avoids multiple traversals of complex terms.

  Arithmetic operations to count the term size are still needed, but the total no. of operations decreases considerably.

- Many different transformations possible.

  Pick one that is cheapest.

Reducing Granularity Control Overheads

- Size-based dynamic granularity control is based on the observation that parallel execution is no longer profitable once input sizes become small enough.
- Usually, argument sizes decrease through a recursion.

  ⇒ Once execution of a procedure becomes sequential, it will stay sequential in subsequent recursive calls.

  Run-time tests for granularity control are no longer needed once execution becomes sequential.

- *Idea* : Use program specialization:
  - Maintain two versions for each (parallel) procedure: one executes in parallel, the other executes sequentially.
  - The sequential version does not have any run-time size tests.

Size Based Granularity Control: Code Generated

```
qsort_G([],[]).
qsort_G([H|L],R) :-
  part_sz(H,L,Sm,Lg,SmSz,LgSz),
  (SmSz >= 10 ->
    (LgSz >= 10 ->
      (qsort_G(Sm,Sm1) & qsort_G(Lg,Lg1)) ;
      (qsort_G(Sm,Sm1) , qsort_S(Lg,Lg1))
    ) ;
    (LgSz >= 10 ->
      (qsort_S(Sm,Sm1) , qsort_G(Lg,Lg1)) ;
      (qsort_S(Sm,Sm1) , qsort_S(Lg,Lg1))
    )
  ),
  append(Sm1,[H|Lg1],R).

qsort_S([],[]).
qsort_S([H|L],R) :-
  part(H,L,Sm,Lg),
  qsort_S(Sm,Sm1), qsort_S(Lg,Lg1),
  append(Sm1,[H|Lg1],R).
```


Ensuring Efficiency of Parallel Execution

- Suppose we want to ensure that parallel execution will not cause a slowdown.

  For a given program p,

  $$T_{par}(p) = max \begin{cases} T_{crit} \\ (T_{seq} + \sum_S freq(S) \cdot overhd(S))/P \end{cases}$$

  where

  S is the set of parallel subtasks;
  P = no. of processors.

- $freq(S)$ can be estimated using any of:
  - structural information, e.g., loop nesting depth [most C compilers];
  - profile feedback [Sarkar 89];
  - cost analysis [Debray & Lin 93].
- Use parallel execution (with size-based dynamic granularity control) only if $T_{par}(p) \leq T_{seq}(p)$.

Estimating Granularity Control Overhead

- For a given program p,

  $$T_{par}(p) = max \begin{cases} T_{crit} \\ (T_{seq} + \sum_S freq(S) \cdot overhd(S))/P \end{cases}$$

  Granularity control overheads can be accounted for within $overhd(S)$.

- Ignoring granularity control overheads can cause a slowdown.

  Example : 16% slowdown on `hanoi(6)` using &-Prolog on a 4-processor Sequent Symmetry.

  19.5% slowdown on `flatten(536)` using ROLOG on a 4-processor Sequent Symmetry.

- *Estimating Granularity Control Overhead* :
  - For any procedure, the overhead at each level of recursion can be estimated from the code generated.
  - Use cost analysis to estimate depth of recursion.
  - Together, they give an estimate of the total granularity control overhead for that procedure.

Lower Bound vs. Upper Bound Cost Estimation

Structure of code with dynamic granularity control:

> **if** ($test(cost\ expr)$) **then**
> execute in parallel
> **else**
> execute sequentially

$cost\ expr$ may involve either lower or upper bound cost estimates.

Upper Bound : if $Cost_{UB}(p) < Threshold$ then:

- Definitely not worth executing in parallel.
- Compile-time estimates are conservative: this means goals are sometimes executed in parallel even when this may not be profitable.
- "Sequentializing a parallel language."

Lower Bound : if $Cost_{LB}(p) > Threshold$ then:

- Definitely worth executing in parallel.
- Compile-time estimates are conservative: this means goals are sometimes executed sequentially even when parallelism is profitable.
- "Parallelizing a sequential language."

Termination Considerations

Upper Bound Cost Estimates :

- Non-terminating computations have cost ∞.

  ⇒ A non-terminating computation will never be sequentialized.

- Granularity control will not affect termination behavior.

Lower Bound Cost Estimates :

- A non-terminating computation may have a finite lower bound.
- Granularity control can sequentialize a non-terminating computation.

  This can adversely affect termination behavior.

Cost Analysis for Or-Parallelism

In Or-parallel execution, the units being parallelized are different branches of the computation tree.

Example : Consider a clause

$$H \ :- \ \ldots, L, L_1, \ldots, L_n.$$

- Suppose L is defined by k clauses $C_i \equiv h_i \ :- \ b_i$, $1 \leq i \leq k$.
- Under or-parallel execution of L, the computation splits into k branches. Branch i consists of:
  - (an instance of) the body b_i; and
  - the continuation $L_1, \ldots, L_n$.
- A cost analysis that does not take continuations into account may underestimate costs.

Continuation Analysis

- Non-tail calls calls "return" to the caller.

  Their continuations are obvious from the call graph of the program.

- A tail call does not return to the caller, but rather to an (arbitrarily distant) ancestor of the caller.

  Such continuations are not obvious from the call graph.

- Construct a context-free grammar as follows:

  For each clause $C \equiv p(\bar{t})\ :-\ q_1(\bar{t}_1), q_2(\bar{t}_2), \ldots, q_n(\bar{t}_n)$ generate a production

  $$p \longrightarrow q_1\ \mathsf{L}_1\ q_2\ \mathsf{L}_2\ \cdots q_{n-1}\ \mathsf{L}_{n-1}\ q_n$$

  where:

  – p, q_i (corresponding to predicate symbols) are nonterminals; and

  – L_i (corresponding to labels) are terminals.

- We use the notion of FOLLOW sets from parsing theory.

  For any procedure p, the set FOLLOW(p) gives the set of labels to which p can return.

  Continuation information is easily recovered from this.

Experimental Results: &-Prolog

[Sequent Symmetry, 4 processors]

Overhead = ε : &-Prolog on a shared memory multiprocessor.

Program	seq	par:ngc	par:gc	δ
fib(19)	1.839	0.729	0.549	24.7%
hanoi(13)	6.309	2.509	2.399	4.4%
matmult	2.099	1.009	0.870	13.8%
qsort(1000)	3.670	1.399	1.409	–0.0%

Overhead = 5ms : representative of an implementation on a distributed memory multicomputer with fast interconnect.

Program	seq	par:ngc	par:gc	δ
fib(19)	1.839	0.970	0.639	34.0%
hanoi(13)	6.309	2.839	2.419	10.1%
matmult	2.099	1.039	0.870	16.3%
qsort(1000)	3.670	1.819	1.429	21.0%

Key :
- seq : sequential speed
- par:ngc parallel speed, no granularity control
- par:gc parallel speed, with granularity control
- δ: (par:ngc – par:gc)/par:ngc

Experimental Results: Muse

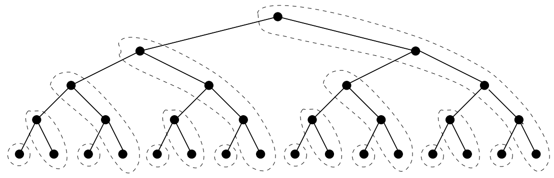
[Sequent Symmetry, 10 processors]

Program	seq	par:ngc	par:gc	δ
queens(8)	17.019	2.090	1.759	15.84%
domino(12)	37.049	4.459	4.139	7.18%
series	22.429	7.360	4.860	33.97%
farmer	17.929	2.170	2.149	0.97%

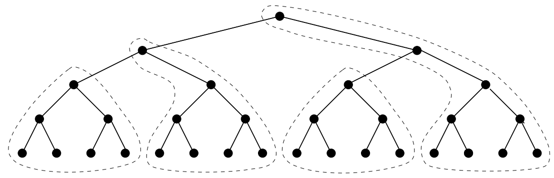
Key :
- seq : sequential speed
- par:ngc parallel speed, no granularity control
- par:gc parallel speed, with granularity control
- δ: (par:ngc – par:gc)/par:ngc

Load-Based Granularity Control

Naive Parallelization :



Breadth-First Until Saturation :



Bread-First Parallelization Until Saturation

Spawn tasks "breadth-first" in the computation tree until the system saturates.

After this, each machine computes sequentially.

Approximations :

Weight-based Scheduling [Tick 88, Tick *et al.* 92] : When an idle processor requests a task, try to find one that is high in the computation tree.

Load-based Inlining : if the system is not loaded, create a separate task; otherwise "inline" and execute sequentially as part of the current task.

Disadvantage : Inlining decisions are irrevocable: once a task has been inlined, this decision cannot be changed.

Lazy Task Creation [Hermenegildo: &-Prolog; Mohr *et al.*: Mul-T]: Busy processors execute their tasks sequentially.

Enough information saved that a processor that becomes idle can steal tasks from a busy processor.

Lazy Task Creation: &-Prolog

Parallel Call Execution : The agent executing the call pushes the arguments on the goal stack, and a PCall frame (containing goal-status info) on its stack.

The parallel call can now be executed by any agent (incl. the one that created it).

Task Stealing : An idle agent searches the stacks of the busy agents looking for a goal to steal.

When a goal is found, the idle agent removes it from the goal stack and starts executing it (after updating status info appropriately).

Remarks :

- When a task is executed by the agent that created it, the overhead is small.
- *However, the overhead is non-zero.* E.g.: shared data structures need to be locked, a marker has to be pushed on the stack, etc.
- With granularity control, these overheads can be avoided for lightweight computations.

Diadora Principle: Massey & Tick 94

- Resembles lazy task creation, but developed for a different reason:
  - Accurate mode analysis in a concurrent logic language is very complex.
  - Massey & Tick propose to use a simpler mode analysis algorithm.
  - However, this analysis may miss some data dependencies between goals.
  - This can lead to deadlock at runtime, due to the sequentialization of a set of tasks with unnoticed circular dependencies.
- *Solution* : When a deadlock is detected, use a mechanism (similar to lazy task creation) to break up the sequentialized task and execute them in parallel.
- *Result* :
  - By default, the system gets the benefits of coarse-grained execution.
  - In the (rare) event of deadlock, lazy task creation is used to recover by splitting a sequential task into finer-grained parallel tasks.