v1.0
BETA

Sistemi Concorrenti e di Rete LS
II Facoltà di Ingegneria - Cesena
a.a 2008/2009

[module 2.4]
ELEMENTS OF CONCURRENT PROGRAMS DESIGN

STEPS IN DESIGN SPACE

- Analyzing the problem to identify exploitable concurrency
  - identifying task / data decomposition and their dependencies
- Choosing a suitable concurrent architecture
  - mapping tasks into agents responsible of task execution
  - shared resources encapsulating result of agent work
  - coordination means to enact and manage dependencies

PROBLEM ANALYSIS

- Objective:
  - identifying task / data decomposition and their dependencies
- Two steps
  1. Task and Data Decomposition Analysis
  2. Dependency Analysis

TASK AND DATA DECOMPOSITION

- *Task decomposition* pattern
  - the problem can be naturally decomposed into tasks
    - then data decomposition follows
  - example
    - “Concurrent quick-sort” problem
- *Data decomposition* pattern
  - problem's data can be easily decomposed into units that can be operated on relatively independently
    - then task related to that units can be identified
  - example
    - “Concurrent Mandelbrot” problem

DEPENDENCY ANALYSIS

- Analyzing dependencies among the tasks within a problem's decomposition
  - grouping and ordering tasks according to the type of dependencies
- Possible kinds of dependencies
  - **temporal** dependencies
  - **data / resource** dependencies
- A classification of dependencies in coordination problem can be found in [MAL-93]
  - interdisciplinary issue

IDENTIFYING CONCEPTUAL CLASSES

- From the analysis we can choose one of three main conceptual classes representing basic methods to solve the problem [CAR-89]
  - **Result parallelism**
    - envisioning parallelism in terms of program’s result
  - **Agenda parallelism**
    - envisioning parallelism in terms of program’s agenda of activities
  - **Specialist parallelism**
    - envisioning parallelism in terms of an ensemble of specialists that collectively constitute the programs

“...To write a parallel program, (1) choose the concept class that is most natural for the problem; (2) write a program using the method that is most natural for that conceptual class; and (3) if the resulting program is not acceptably efficient, transform it methodically into a more efficient version by switching from a more-natural method to a more-efficient one” [CAR-89]

“BUILDING A HOUSE” EXAMPLE (1/3)

- Adopting a *result parallelism* approach
  - think about the components of the house
    - front, rear, side walls, interior walls, foundations, roof, etc.
  - proceed by building all the components simultaneously
    - assembling them as they are completed
  - separate workers set to work on the different parts, proceeding in parallel up to the point where work on one component can’t proceed untile another is finished

“BUILDING A HOUSE” EXAMPLE (2/3)

- Adopting a *specialist* approach
  - identify the separate skills required to build the house
    - surveyors, excavators, foundation builders, carpenters, roofers, etc
  - then assemble a constructor *crew* in which each skill is represented by a separated specialist worker
  - they all start simultaneously
    - but initially most workers will have to wait around
    - once the project is well underway, however, many skills (hence many workers) will be called to play simultaneously
  - strong role of worker interaction and coordination
    - different kind of strategies
    - e.g. pipelined jobs
- Opposite approach with respect to *result parallelism*

“BUILDING A HOUSE” EXAMPLE (3/3)

- Adopting an *activity agenda* approach
  - we write out a sequential agenda and carry it in order, but at each stage we assign many workers to the current activity
    - doing the foundation, doing the frame, doing the roof, etc
  - we assemble a work team of *generalists*
    - each capable of performing any construction step
  - the team is set to work stage by stage following the agenda

RESULT PARALLELISM

- Designing the system around the data structure or resource yielded as the ultimate result
  - we get parallelism by computing all the elements of the result simultaneously
- Each agent is assigned to produce one piece of the result
  - they all work in parallel up to the natural restriction imposed by the problem
- Proper shared data structures are designed to wrap the result (data) under construction
  - encapsulating mutex and synchronization issues

SPECIALIST PARALLELISM

- Designing the system around an ensamble of specialists connected into a logical network of some kind
  - parallelism results from all the nodes of the logical network (all the specialists) being active simultaneously
- Each agent is assigned to perform one specified kind of work
  - they all work in parallel up to the natural restriction imposed by the problem
- Coordination means (communication protocols, shared data structures) are used to support specialists communication and coordination
  - examples: message boxes, blackboards, event services

AGENDA PARALLELISM

- Designing the system around a particular agenda of activities and then assign many workers to each step
- Each agent is assigned to help out with the current item on the agenda
  - they all work in parallel up to the natural restriction imposed by the problem
- Shared data structures are designed to manage data consumed and produced by agenda activities
  - examples: bounded-buffers

REMARKS

- The boundaries between the three classes are not rigid
  - we will often mix elements of several approaches in getting a particular job done
    - e.g. a specialist approach might make secondary use of agenda parallelism, for example, by assigning a team of workers to some speciality
- However the three approaches represent *three clearly separate ways of thinking about the problem*
  - result parallelism focuses *on the shape of the finished product*
  - specialist parallelism focuses *on the makeup of the work crew*
  - agenda parallelism focuses *on the list of tasks to be performed*
- The approach can be recursively applied
  - following task and data decomposition

FROM CONCEPTUAL CLASSES TO ARCHITECTURES

- Some specific architectures can help to bridge the gap between the conceptual class and the implementation [MAG-99]
  - Master-Workers
  - Filter-Pipeline
  - Announcer-Listeners

MASTER-WORKERS

- Also called *Supervisor-Workers* or *Manager-Workers*
- Description
  - the architecture is composed by a *master* agent and a possibly dynamic set of *worker* agents, interacting through a proper coordination medium functioning as a *bag of tasks*
  - **master agent**
    - decompose the global task in subtasks
    - assign the subtasks by inserting their description in the bag
    - collect tasks result
  - **worker agent**
    - get the task to do from the bag, execute the tasks and communicate the results
  - **bag of tasks resource**
    - typically implemented as a blackboard or a bounded buffer
- Example of problems
  - quadrature problem
  - quick sort

FILTER-PIPELINE

- Description
  - the architecture is composed by a linear chain of agents interacting through some *pipe* or *bounded buffer* or *channel* resources
  - **generator agent**
    - the agent starting the chain, generating the data to be processed by the pipeline
  - **filter agent**
    - an intermediate agent of the chain, consuming input information from a pipe and producing information into another pipe
  - **sink agent**
    - the agent terminating the chain, collecting the results
- Example of problems
  - Primes Sieve

ANNOUNCER-LISTENERS

- Description
  - the architecture is composed by an *announcer* agent and a dynamic set of *listener* agents, interacting through an *event-service*
  - **announcer agent**
    - announce the occurrence of events on the event service
  - **listener agents**
    - register on the event service so as to be notified of the occurrence of events interesting for the listener
  - **event-service resource**
    - uncouple announcer-listeners interaction
    - collect and dispatch events
- Example of problems
  - Structuring GUI in concurrent programs

REVISITING THE MODEL VIEW CONTROLLER

- Issue
  - how to separate model / view / controller aspects in the design of complex concurrent programs
- A mapping
  - encapsulating control in agents
    - active components of the program
    - encapsulating the control logic
  - model as passive shared data structure, enforcing mutex
    - can be used by agents
    - encapsulating mutex properties
  - view as passive shared data structures, with synch functionalities
    - enabling and mediating the interaction with users
    - observed and used by agents
    - can access to model resources to get the state

BIBLIOGRAPHY

- [CAR-89]
  - Nicholas Carriero and David Gelernter, "How to write parallel programs: a guide to the perplexed", ACM Computing Surveys (CSUR). Volume 21, Issue 3 (1989)
- [MAT-05]
  - T. Mattson, B. Sanders, B. Massingill, "Patterns for Parallel Programming", Addison Wesley (2005)
- [MAG-99]
  - Jeff Magee and Jeff Kramer, "Concurrency - State Models and Java Programs", Wiley (http://www.doc.ic.ac.uk/~jnm/book/)
- [MAL-93]
  - Thomas W. Malone, Kevin Crowston. "The interdisciplinary study of coordination". Center for Coordination Science, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, 1993