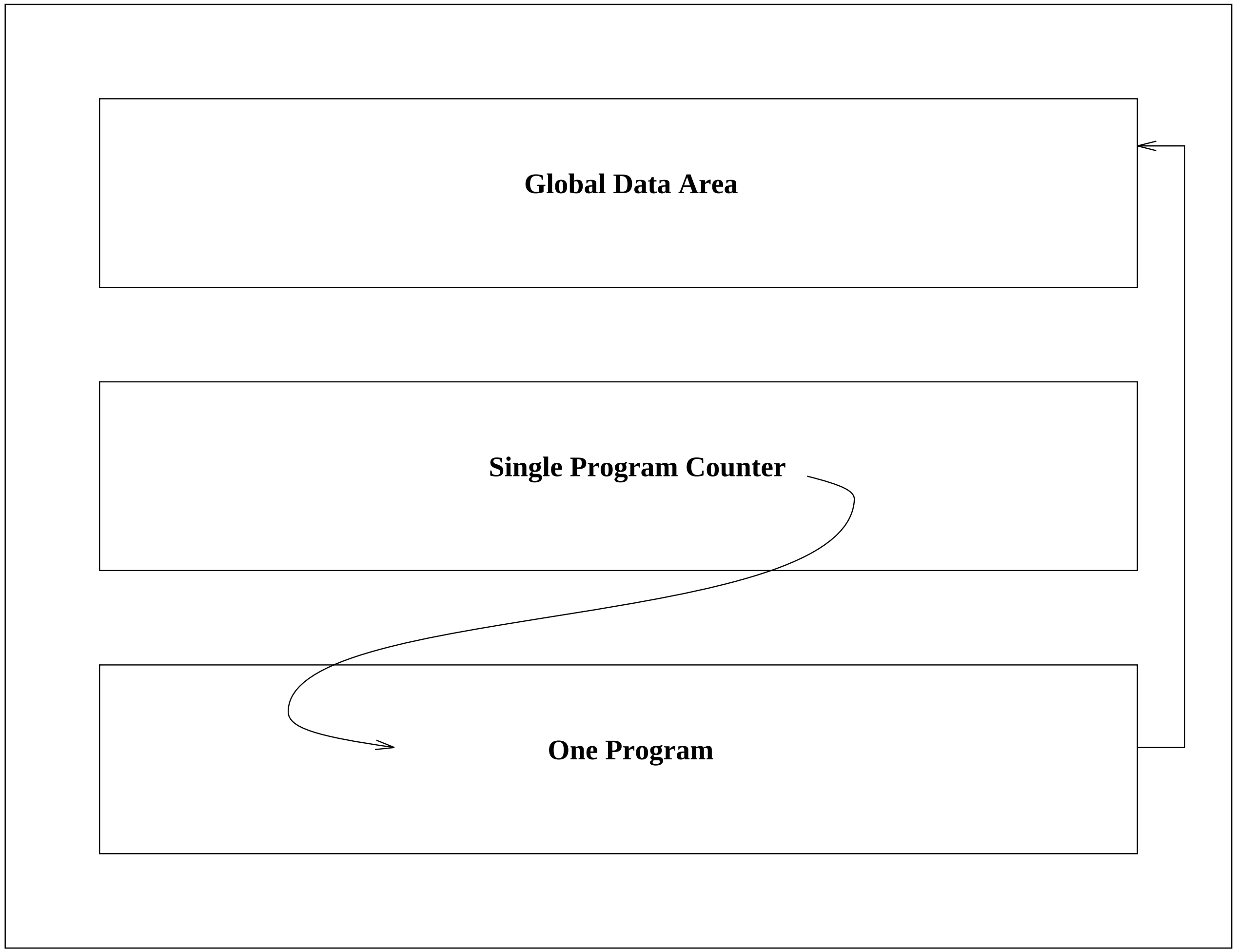
Efficient Implementation of High Performance Fortran via Adaptive Scheduling–An Overview

Edward A. Kornkven
L.V. Kalé
University of Illinois

Data Parallel Programming



Data Parallel Programming

- "Like SIMD"
- Single thread of control
- Global name space
- Loosely synchronous execution
- Relatively easy to program
  - Similar to programming a uniprocessor
  - Familiar approach to programming
  - Minimal amount of interface to underlying machine
  - Race conditions are not possible

The DP-Charm Language

- The data parallel source language
- Based on High Performance Fortran (HPF)
- Array-oriented
- Array data is distributed across processors
  - Many processors operate on an array $\Longrightarrow$ parallelism
  - User specifies distribution of each dimension
- Many array operators and intrinsics

Sample HPF Source Code

```
      DIMENSION A(50), Anew(50)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK), Anew(BLOCK,BLOCK)
      A = Anew
      Anew = (1/(2+2*c)) * (CSHIFT(A,-1,1) +
             CSHIFT(A,1,1) +
             c*(CSHIFT(A,-1,2) + CSHIFT(A,1,2)) - W)
```

A(50) $\longrightarrow$ BLOCK

PE:	1	2	3	4	5	6	7	8	9	10
	1	6	11	16	21	26	31	36	41	46
	2	7	12	17	22	27	32	37	42	47
	3	8	13	18	23	28	33	38	43	48
	4	9	14	19	24	29	34	39	44	49
	5	10	15	20	25	30	35	40	45	50

B(50) $\longrightarrow$ CYCLIC(3)

PE: 1	2	3	4	5	6	7	8	9	10
1	4	7	10	13	16	19	22	25	28
2	5	8	11	14	17	20	23	26	29
3	6	9	12	15	18	21	24	27	30
31	34	37	40	43	46	49			
32	35	38	41	44	47	50			
33	36	39	42	45	48				

Data Parallelism $\longrightarrow$ MIMD

The Traditional Approach

- Simulate global name space
  - Distribute array data across processors
  - Each processor operates on its piece
  - Generate communication when necessary
- Provide loosely synchronous execution
  - SPMD-style execution
- Maintain a single thread of control
  - All processors are at approximately the same place in the execution

Insight

- Although the semantics is single-threaded, the implementation doesn't have to be
- The implementation needs to be constrained only by dependences
  - In particular, dependences between messages and computations

Insight

- Although the semantics is single-threaded, the implementation doesn't have to be
- The implementation needs to be constrained only by dependences
  - In particular, dependences between messages and computations

```
Compute A
x = SUM(A)
y = MAX(A)
B = A/x
C = A/y
D = B + C
```

Data Parallelism $\longrightarrow$ MIMD

The DP-Charm Approach

- Simulate global name space
- *Simulate* loosely synchronous execution
  - Actually based on message-driven execution
- *Simulate* single thread of control
  - Actual executable has multiple threads of control

Dynamic Adaptive Schedules

- Generation of dynamic adaptive schedules $\Longrightarrow$ Compiler
  - To enable scheduling alternatives and adaptability
- Message-driven execution $\Longrightarrow$ Charm
  - To implement multi-threading
- Macro-dataflow $\Longrightarrow$ Dagger
  - Message-driven and multi-threaded execution
  - Program tasks are partially ordered according to necessary dependences

Charm

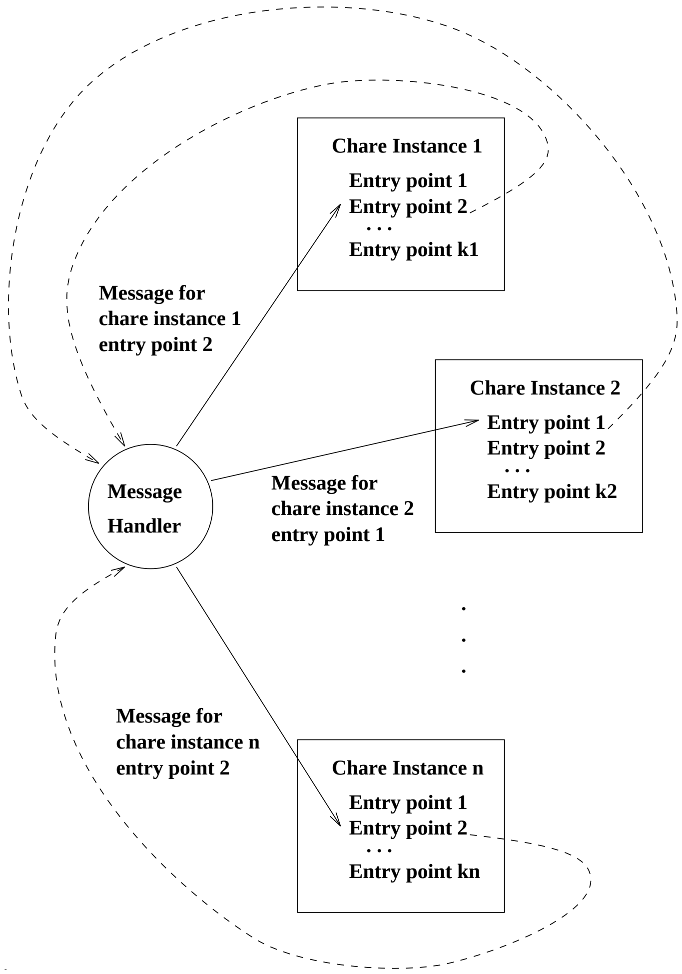
- Computations in Charm are composed of objects called chares.
- Chares are instantiated under user control.
- A chare instance contains data and methods called *entry points* to which messages may be directed.
- Entry points run to completion at which time the chare suspends until given another message.
- A chare instance on every processor is maintained by a *branch-office chare* (BOC).
- There is no *receive* statement—chares execute in a message-driven fashion.

Dagger Notation

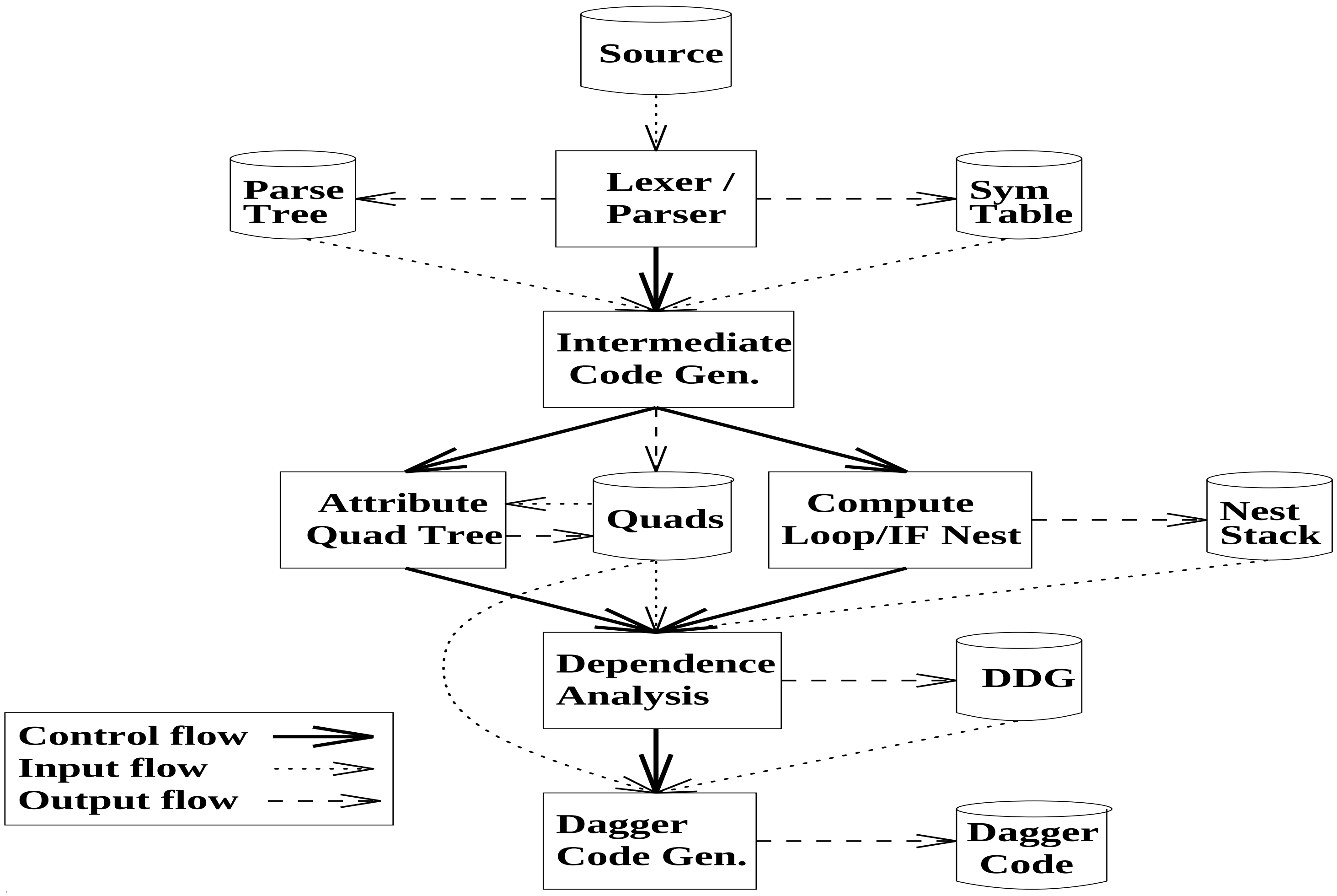
- Built on top of Charm
- `when` blocks: Charm entry points guarded by
  - Message entry points
  - Condition variables
- `when` block is enabled for scheduling when all dependences have been fulfilled

when $cv_1, cv_2, cv_3, \ldots, cv_m, e_1, e_2, e_3, \ldots, e_n$:
$\{$ *code to be executed* $\}$

Message-driven Execution & Charm



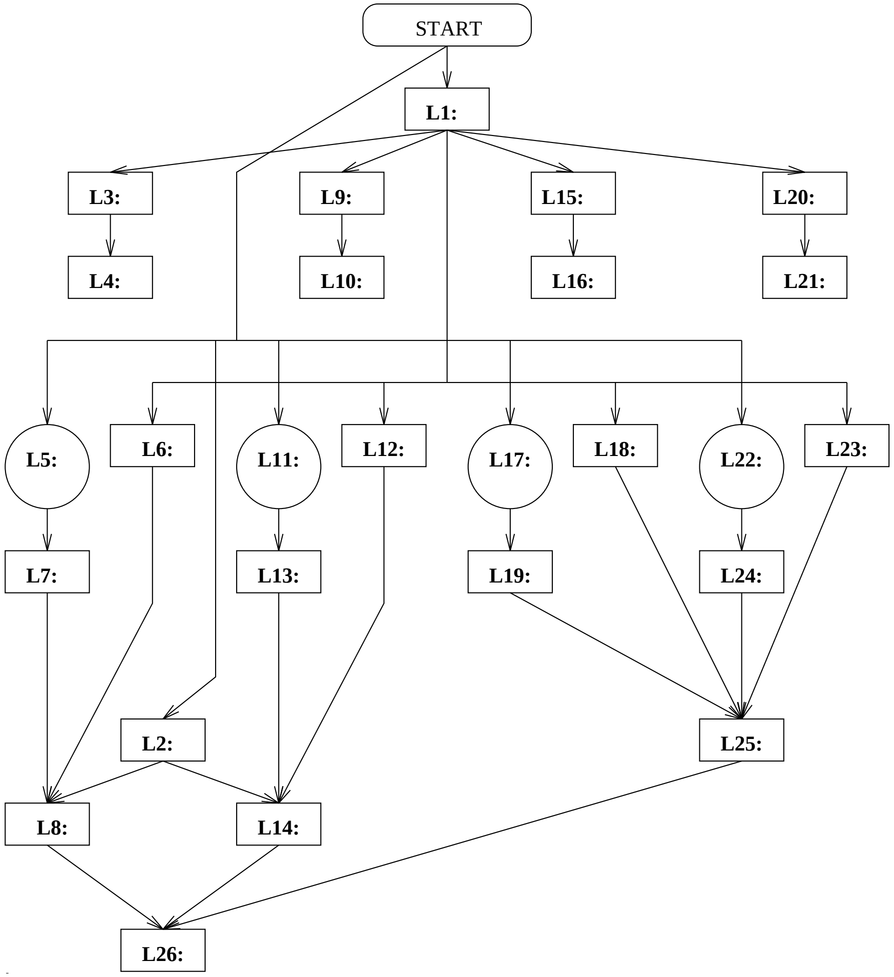
A Compiler for DP-Charm



Data Dependences

- Captured in the *data dependence graph* (DDG)
- Two kinds of nodes
  - rectangles: computations
  - circles: messages
- Edges between rectangles represent dependences that must be preserved
- Dependences imply a partial order
  - Source language over-specifies order (gives a total order)
  - DDG gives only necessary partial order

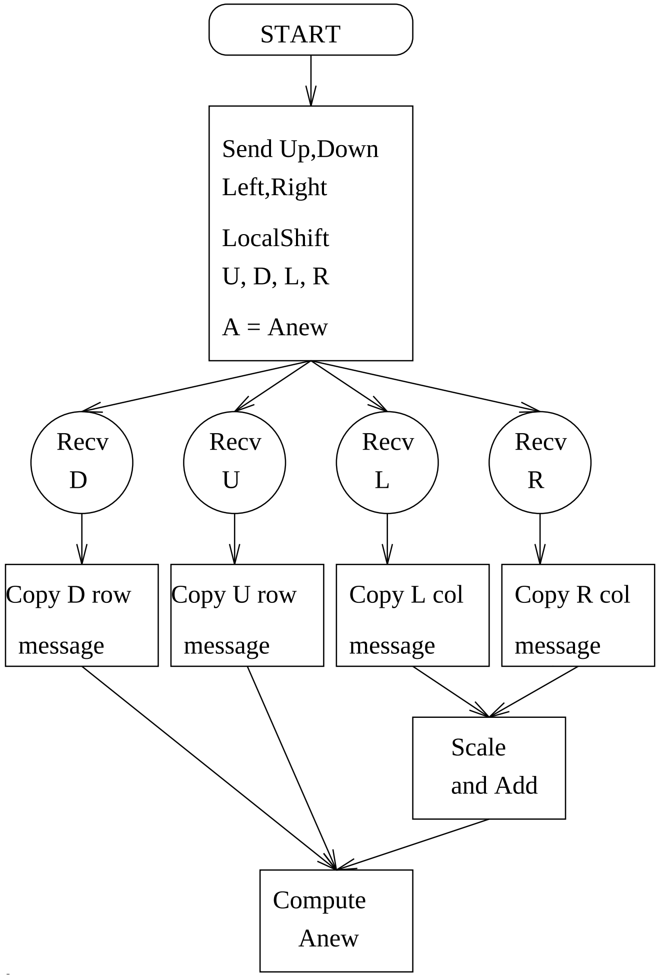
The Initial DDG



DDG Transformations

- Each node of the DDG will become a Dagger `when` block.
- Each Dagger `when` block is separately scheduled at run time.
- $\Longrightarrow$ fewer `when` blocks will reduce overhead.
- $\Longrightarrow$ we would like to reduce the number of DDG nodes.
  - **Node merging**
    - if a node has a single predecessor, combine them.
  - **Redundant arc removal**
    - may allow further node merging

The DDG After Transformations



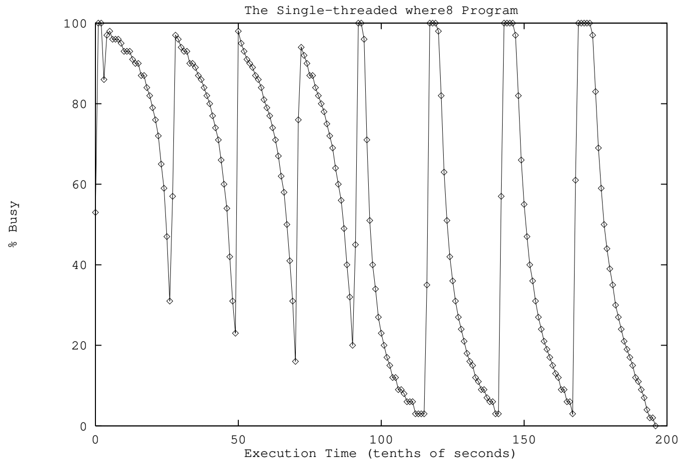
Translation of DP-Charm Assignments

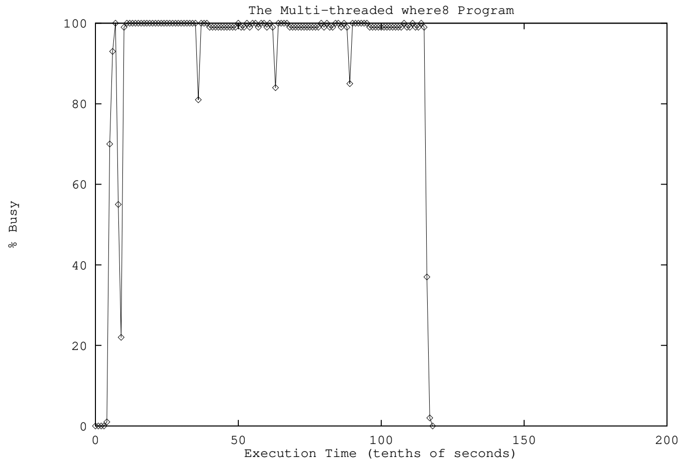
LHS	**RHS**			
	Array	**Section**	**Element**	**Scalar**
Array	copy/ redist	eltwise send	broadcast	copy
Section	eltwise send	eltwise send	eltwise send	eltwise send
Element	illegal	eltwise send	eltwise send	copy on 1 PE
Scalar	illegal	broadcast	broadcast	copy

Handling While loops and Conditionals

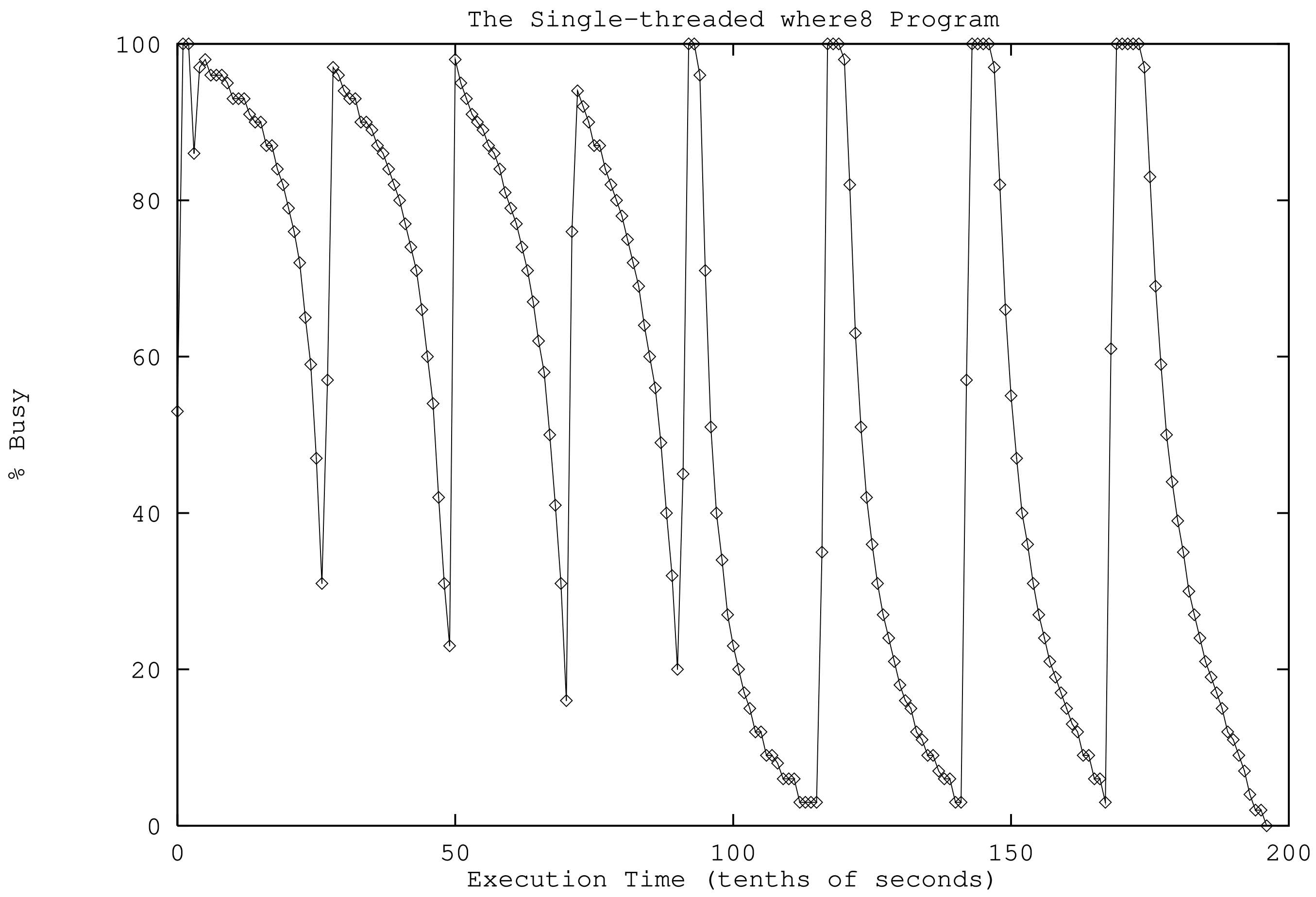
- Introduce complexity
- Enforce dependences within loop bodies
- Enforce dependences across loops
- Handle nesting
- Handle out-of-order messages

Single-threaded vs. Multi-threaded Performance

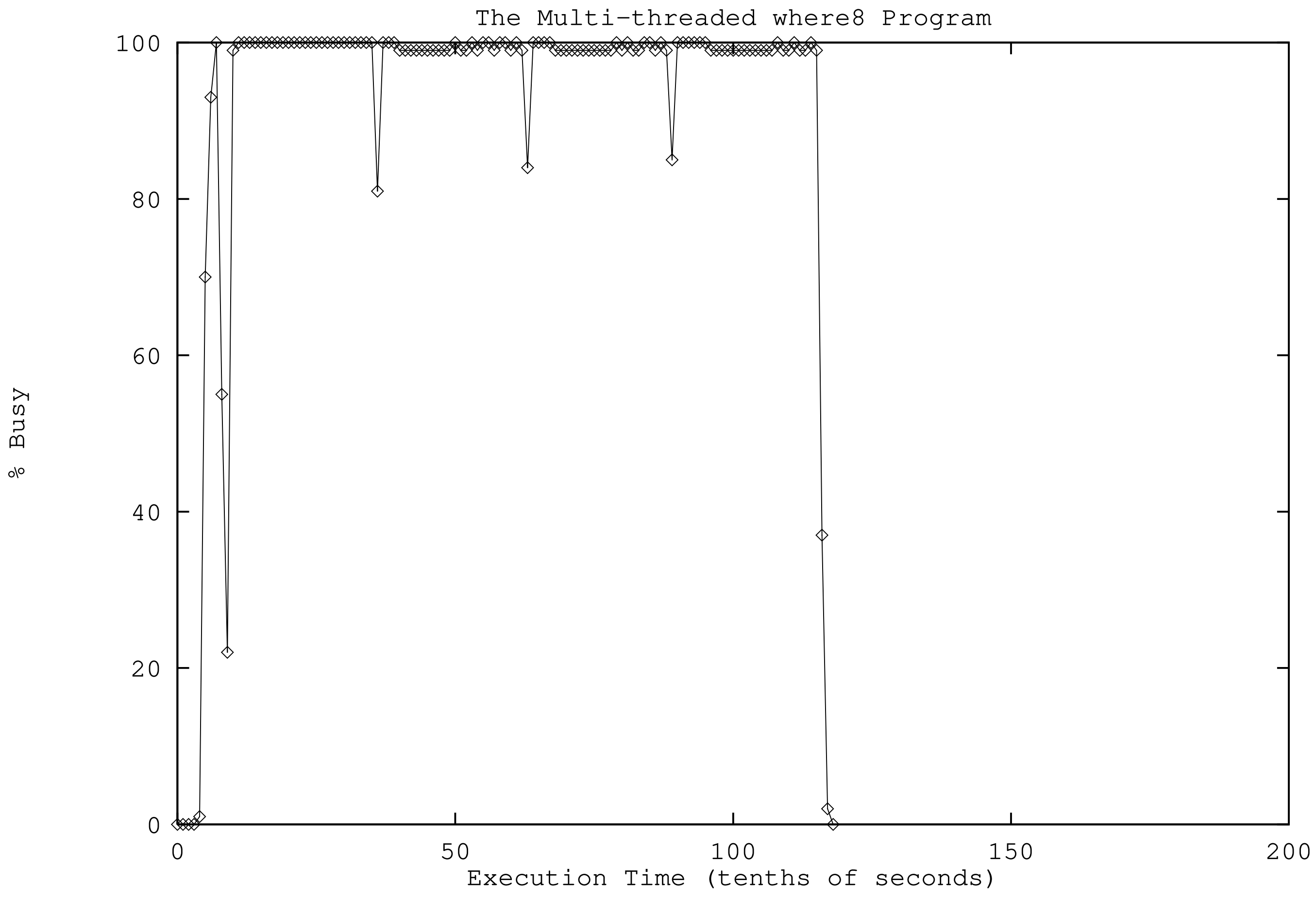




Single-threaded Performance



Multi-threaded Performance



Array Representation

- Storage - arrays, scalars
- Base types: INTEGER, REAL, LOGICAL, CHARACTER
- Array descriptor

Name
User data
Data type
Number of elements
Element size
Number of dimensions
Dimension information

Dimension Descriptor

Distribution method
Distribution specification
Number of elements
Lower and upper index bounds for this segment
Lower and upper index bounds for user array
Number of processors
Processor number on this dimension

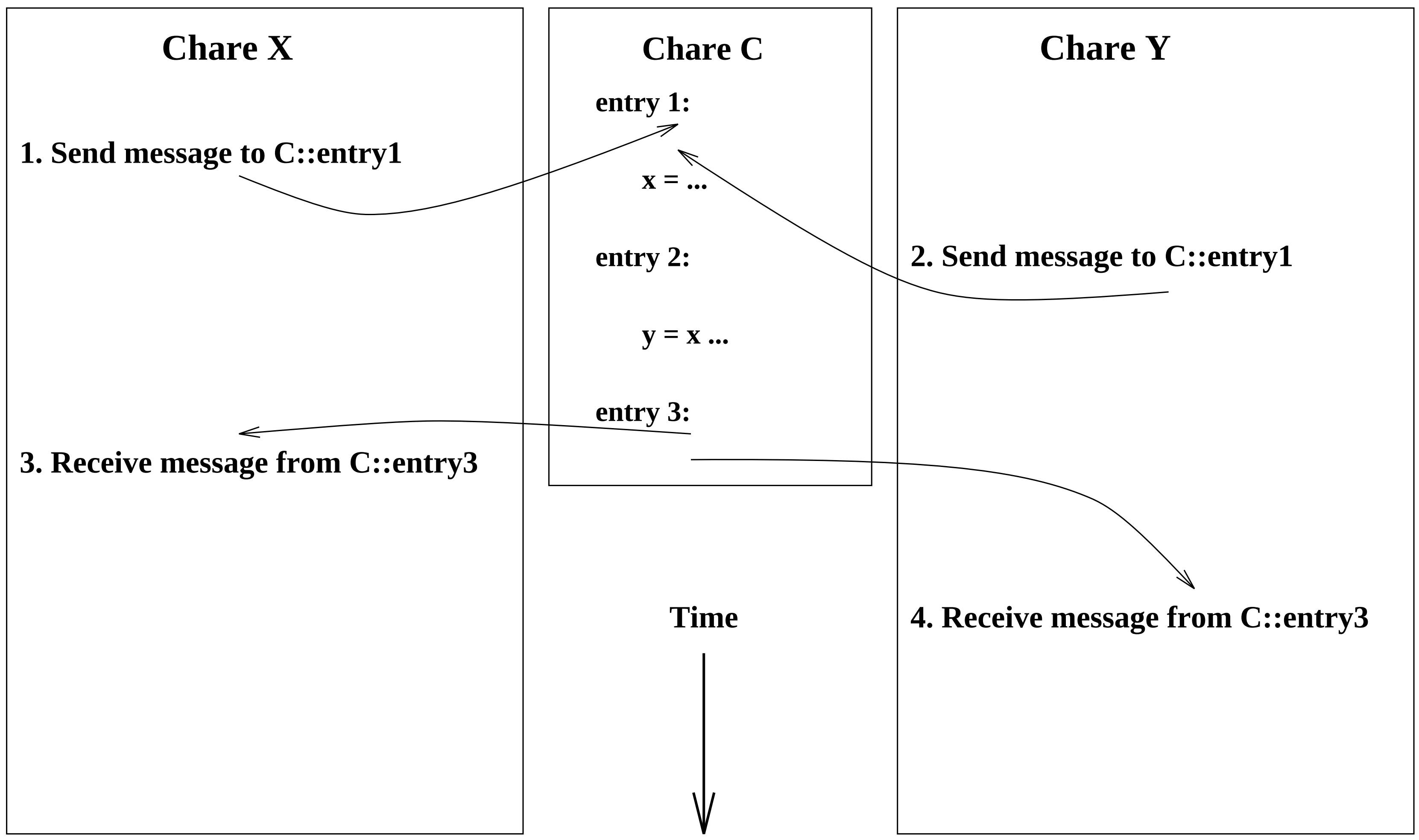
Sample Control Nest Stack

Statement List	Pointer	Nest Stack
A = 0		**(empty)**
do while (c1)		**do while (c1)**
B = B + 1		**do while (c1)**
if (c2) then		**if (c2) → do while (c1)**
C = C + A		**if (c2) → do while (c1)**
end if		**do while (c1)**
end do		**(empty)**

Code Sharing

- Dagger programs are *split phase*.
- Between `when` blocks, control may be passed to another chare/entry point.
- Therefore, a chare may be invoked at an empty point before a former invocation has finished.
- Must be able to share that block of code between the two callers.
- Solution:
  - Distinguish different invocations via *reference numbers*
  - Maintain copies of local state outside the shared code

Example of Code Sharing



DDG Construction