Graphical/Visual Abstract Models and Tools in Parallel Computation

Tutorial

Supercomputing ‘93

J.C. Browne

Peter Newton

University of Texas at Austin

Dept. of Computer Sciences

Contents

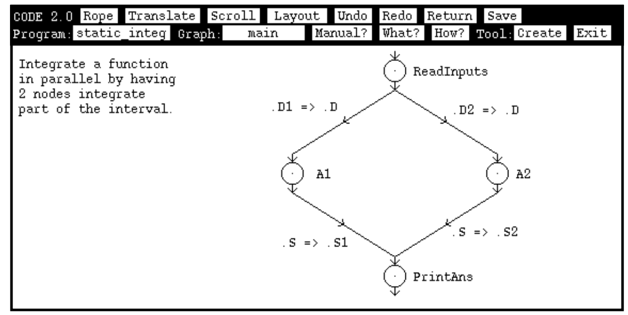
- Brief Introduction to CODE 2.0 Parallel Programming Environment.
- Advantages of Abstract Models.
- Advantages of “task” models.
- Composition in Abstract Models.
- Some Graph-Based Models.
- CODE 2.0 Model and Environment.
- CODE 2.0 Example.
- CODE 2.0 Lab Exercise.

Introduction to CODE 2.0

- Example of realized graphical/visual model.
- System goals:
    - Easy of use.
    - Expressiveness.
    - Portable program representation basis.
    - Efficient executable structures.
- Goals tend to exclude each other. Many systems address only one.
- CODE 2.0: *balanced* approach to all goals.

Informal System Basis

- Example: Parallel Integration.



- Draw picture of parallel structure.
- Structure = graph showing communication patterns.
- Annotate.
- UC node is basic primitive.
- Translate for selected target architecture.
- Also supports multiple graphs, runtime determined graph structures, shared variables, etc.

ReadInputs Spec

```
output_ports { IntegInfo D1; IntegInfo D2; }

vars { real a; real b; real h; int NumPoint;
      IntegInfo d1; IntegInfo d2; }

comp {
  a = ReadReal(); b = ReadReal();
  NumPoint = ReadInt();
  h = (b - a)/2;
  d1.a = a; d1.b = h; d1.n = NumPoint/2;
  d2.a = h; d2.b = b; d2.n = NumPoint/2;
}

routing_rules {
  TRUE => D1 <- d1; D2 <- d2;
}
```


A Spec

```
input_ports { IntegInfo D; }

output_ports { real S; }

vars { IntegInfo d; real val; }

firing_rules {
  D -> d =>
}

comp {
  val = simp(d.a, d.b, d.n);  // Integrate using Simpson's rule.
}

routing_rules {
  TRUE => S <- val;
}
```

PrintAns Spec

```
input_ports { real S1; real S2; }

vars { real s1; real s2; }

firing_rules {
  S1 -> s1, S2 -> s2 =>
}

comp {
  PrintReal(s1+s2);
}
```


Running the Program

```
pageboy-25 cd static_integ.serial
pageboy-26 ls
Makefile        c2_globtype.h c2_main.c       c2_main.h
   main.c
pageboy-27 make
/bin/cc -g -I/v/peaches/v1/code2/code/serial/code -c
   c2_main.c
/bin/cc -g -I/v/peaches/v1/code2/code/serial/code -c main.c
/bin/cc c2_main.o  main.o .../c2libo.o  -o static_integ -lm
pageboy-28 static_integ
Enter an double: 0.0
Enter an double: 1.0
Enter an integer: 1000
The real = 3.141593
 _c2_StopAll called to end computation
Worker 0: Total 5, Idle 0, Runs 5, Reruns 0
Worker 0 terminates
```

Abstract Model of Parallel Computation

- **Most parallel programming currently based on ad hoc models.**
  - **Message passing libraries.**
  - **Extensions to sequential languages.**
- **Appropriate abstract models facilitate program creation process.**
  - **Emphasize <u>important</u> aspects of problem domain.**
  - **Hide unimportant aspects.**
  - **Free programmer for mentally mapping from informal model to runtime model.**
- **Model use can vary.**
  - **Good: design phase.**
  - **Better: design phase + implementation phase.**
  - **Best: design + implementation + debug + performance analysis.**

What is the Model Problem?

- **One model for programming.**

```
void a () {
  while (NotDone) {
      *
      *
      send(b, mess);
      recv(c, message);
      *
      *
  }
}
```

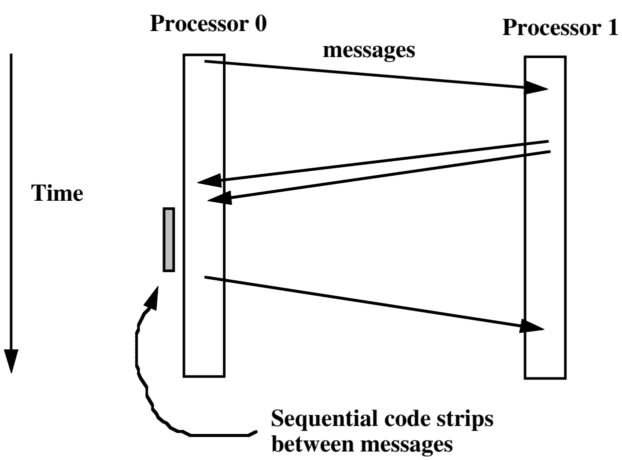
```
void b () {
  while (NotDone) {
      *
      *
      if (w)
        send(b, mess);
      else
        send(a, mess);
      recv(c, mess);
      *
      *
  }
}
```

```
void c () {
  while (NotDone) {
      *
      *
      send(a, mess);
      send(b, mess);
      recv(b, message);
      *
      *
  }
}
```

```
void main () {
  spawn(a);
  spawn(b);
  spawn(c);
}
```

The Model Problem?

- Another model for execution.



What is the Connection?

- Programming model => runtime model.
  - In what state (control flow) can a certain message be sent?
  - Can there be deadlock?
  - What is the granularity (time between messages)?
  - Sending message to "near" processor or "far"?
- Runtime model => programming model
  - What process sent this message?
  - What send in the process sent the message?
  - How did that process reach that state?
  - This processor is very busy and other are not. Why?

Environments Based on Abstract Models

- **Eliminate the model mapping problem.**
- **Programmers think in only one model.**
    - **Machine is abstract.**
    - **Programming model = runtime model.**
- **Must choose model carefully.**
    - **Good for programmer thought process.**
    - **Easy to automatically map to real machine in manner that preserves performance.**
    - **Optimization in abstract model must also optimize in real model.**
- **Key to programming parallel machines: find the right model.**
    - **Need not serve all, but**
    - **Apply to significant domains**
    - **Apply to large architecture classes.**

One Model is Better than Two

- **Choose one unified model for both programming and runtime representation.**
- **CODE 2 model proposed.**
- **Superior programming abstractions.**
- **Direct understanding of runtime behavior.**

The TASK-DEPENDENCE Model of Parallel Computation

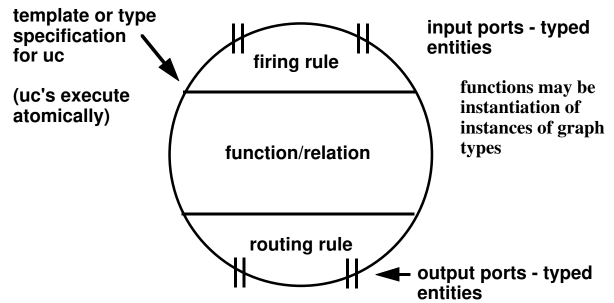
Definitions

- **A task is a unit of computation which, once initiated, executes atomically.**
- **A dependence is an input/output relation between task or a name sharing relation among tasks.**
- **A program is a set of tasks and a specification of the dependences among that set of tasks.**
- **An execution of a program is an sequence of instances of execution of the tasks ordered by the dependences among the task instances.**

- **Properties**
  - **Program and execution models are directly related which facilities design for performance and debugging.**
  - **Task specifications are component interfaces which promotes component reuse.**

Task Definition

- **A task is a unit of computation which includes:**
  - **an initial state.**
  - **a firing rule - a specification of the states which enable execution of the function or relation.**
  - **a function or relation with a signature.**
  - **a routing rule - a specification for disposition of the output of the function or relation.**

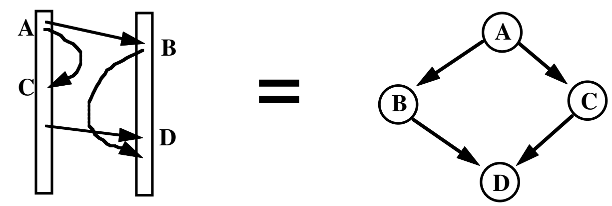


Compositional Programming

- Compositional Programming is a mode of thinking about programming. In the compositional model:
- "A parallel program is a set of tasks composed into a structure by composition operators which define relationships among tasks. "
- The composition operators in the task/dependence model of parallel computation are dependences:
  - (a) data dependencies which map from output ports to input ports.
  - (b) name sharing dependences which specify access to shared names.
- Compositional programming with the task model is our proposal for the "right' model of parallel programming.

Graph-Based Models Are Natural

- Parallel Computations are 2-dimensional. Graphs directly represent them.



- Graphs are a natural design tool, regardless of implementation language.
- Many programmers draw process diagram already as part of design process.
- Just continue in that notation.

Graph-Based Models Separate Parallel Structure from Sequential Function

- **Graphs show parallel structure.**
- **Sequential functions encapsulated in nodes or arcs, depending on nature of the graph model.**
- **Large advantage.**
  - **Graphs good at showing structure.**
  - **Text good for sequential programming.**
  - **Use each for its best purpose.**
- **Do not re-invent sequential programming-- embed it.**
  - **Use existing tools, languages, and techniques.**
  - **Reuse old programs or fragments.**
  - **Reuse old libraries of program components.**
  - **Don't make programmers learn too many new tricks at once. (They will, perhaps wisely, refuse!)**

Graphs and Performance

- **Graph notations can make program structure <u>explicit</u>.**
  - **Users understand granularity.**
  - **Users understand locality issues.**
- **Non-shared name graph models.**
  - **Encourage large grain by encapsulating computation.**
  - **Encourage good locality for same reason.**
  - **Encourage clean design by forcing clean interfaces.**
  - **Experience in UT parallel programming classes.**
- **Explicit representation simplify compilers.**
  - **No need to discover parallelism.**
  - **Limited name ambiguities (most graph models).**
  - **Do need to exploit parallelism intelligently.**

A Note on Flowcharts

- **Graphical techniques less compelling for sequential programming.**
- **Graphs good at showing large-scale structure .**
  - **Important for performance.**
  - **Component isolation.**
- **Flowcharts show large-scale (linear) control-flow.**
  - **Not needed with good sequential programming practice.**
  - **Program hierarchically structured into blocks.**
  - **Control-flow is trivial within blocks.**
  - **Limited performance concerns.**
- **Graphical techniques can be good for showing modularity relationships: call trees, class browsers, etc.**
  - **Also useful in parallel programming.**

Some Graph Models which have been used for representation of parallel programs

- **Petri Nets.**
- **Data Flow Graphs.**
- **State Transition Graphs.**
- **Karp-Miller Computation Graphs.**
- **etc.**

in fact,

1. **All interesting parallel computation structures (programs) can be represented as directed graphs with appropriate choices of semantics for the the nodes and arcs.**

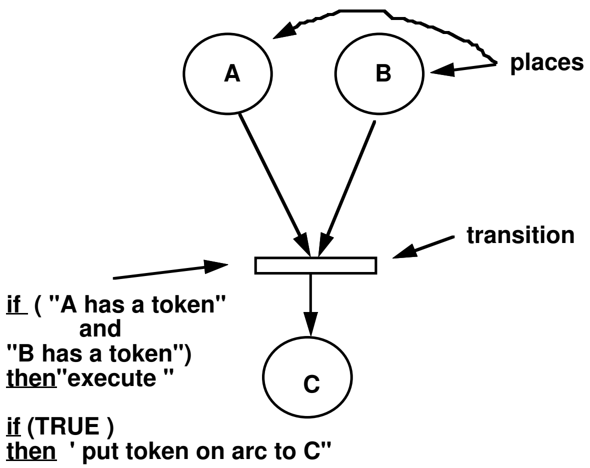
2. **All execution of these parallel programs can be represented as traversals of these graph structures.**

so,

let us look at a few of the historically interesting models in this context.

Petri Nets

- Nodes types are Places and Transitions.
- Arcs bind instances of node types.
- Tokens are state variables.

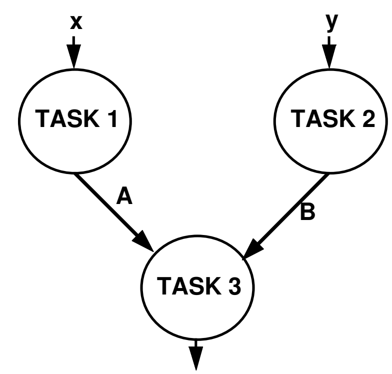


place = data flow arc

transition = simple unit of computation

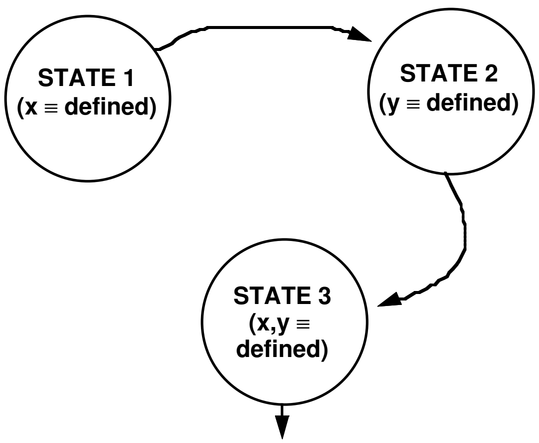
Dataflow Graphs

- Nodes are tasks.
- Arcs are buffers for values.
- Firing rules are implicit. Task 3 can execute when values from "A" and "B" are present on the connecting arcs.
- Tasks are “black boxes.”
- routing rules are also implicit.
- State of program specified by data values on arcs of data flow graphs.



State Transition Graphs

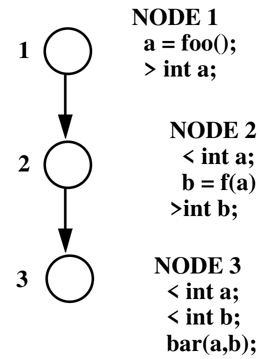
- Name/value bindings at nodes.
- Arcs carry functions which are executed.
- when a node attains a given state.



IF (STATE = STATE 1) THEN EXECUTE "Task 1" (which leads to state 2, etc.)

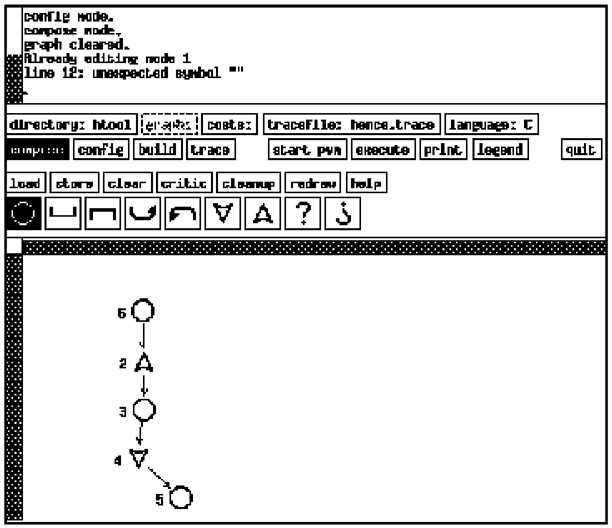
HeNCE Graphs

- Nodes are functions .
- Dataflow is not graphical, but ordering is determined by arcs.



- Node 3 gets "a" from node 1 and "b" from node 2.

HeNCE Environment

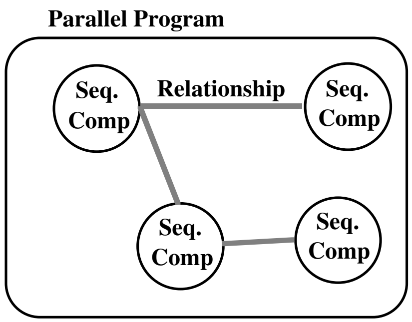


- Program shows “fan” construct.
- Produces PVM program.

The CODE 2.0 Model and Environment

- First discuss sequential programming.
  - External sequential functions.
  - UC node (basic component) specification.
- Then discuss parallel programming.
  - Composition of UC nodes into graphs.
  - Composition of graphs into programs.

Compositional Programming in CODE



- Sequential computations– black boxes, atomically transform inputs to outputs.
- Do not re-invent sequential programming.
- CODE *requires* rigorous interfaces.
- CODE focus on abstractions for *parallel* programming.

Types and External Functions

- Types-- int, real, char are built in.
- Named types, arrays and structure types-- named in program attribute form.

```
type Mat1 is array of real;  // note: no bounds!
type Mat2 is array of Mat1;
type Point is struct {
  real x;
  real y;
}
```

- External functions-- specify interface in program attribute form.

```
void PrintReal(real x);
     // note "*" for passing ints, reals, or chars
     // by reference.
int foo(Mat2 m, Point SomePoint, int *n);
```

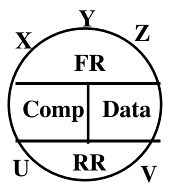
CODE Sequential Computations

- **Presentation: focus on primitive unit and then on composition.**

- **Essential idea– the basic Unit of Computation.**

Basic Unit:

UC Node



Ports define interface

- **Firing Rules– when computation can fire.**
- **Comp– the sequential computation.**
- **Data– state may be retained.**
- **Routing Rules– where to send outputs.**
- **Textual specification used in CODE.**

Example Unit of Computation

```
input_ports { int X; int Y;}
output_ports { int U; }
shared_vars { Mat1 M reader; }

vars { int v; int Done; int w; }

init_comp {
  Done = 0;
}

firing_rules {
  X -> v, Y -> w =>
}
comp {  // note calls to external funcs.
  while (!Done) {
    if (v > 0) DoSomething(v, Done, M);
    else DoSomethingElse(v, M, w);
  }
}
routing_rules {
  TRUE => U <- v;
}
```

Firing Rules Stanza

- More than one rule-- non-determinism.
- Rules must be “or-ed”.

```
firing_rules {
  X -> v => ||
  Y -> v =>
}
```

- Local variables on guards and bindings

```
firing_rules {
  state == 0, X -> v => state = 1;  ||
  state == 1, Y -> v =>
}
```

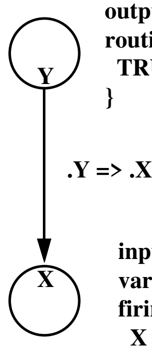

Routing Rules Stanza

- Multiple rules must be “and-ed”.
- Can use local variables.
- If no guard TRUE, no bindings are done.

```
routing_rules {
  Even(n)  => X <- v; state = 1; &&
  Odd(n), state == 0  => Y <- v; X <- v;
}
```

Composition into Parallel Programs

- Draw arcs between Unit of Computation Nodes.
- Arcs simply bind names, either ports (for dataflow) or shared variables.



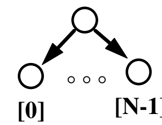

```
output_ports { int Y; }
routing_rules {
  TRUE => Y <- 7;
}
```

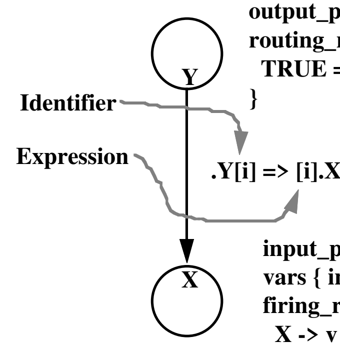
```
input_ports { int X; }
vars { int v; }
firing_rules {
  X -> v =>;
}
comp { foo(v); }
```


More Complex Composition

- Runtime-determined structure

One node sends to all.
(could be subset)



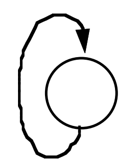



```
output_ports { int Y; }
routing_rules {
  TRUE => { Y[i]  <- m[i]; : (i N) };
}
```

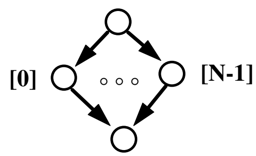
- Arc determines UC replication.

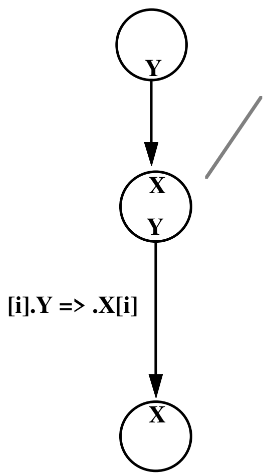
```
input_ports { int X; }
vars { int v; }
firing_rules {
  X -> v =>
}
comp { foo(v); }
```

- Powerful notation. Efficient results.
- Self-loop may expand into any graph structure.



Gathering Values into an Array






```
input_ports { int X; }
vars { int v; }
firing_rules {
  X -> v =>
}
comp { foo(v); }
routing_rules {
  TRUE => Y <- v;
}
```

```
input_ports { int X; }
vars {Mat1 v[N]; }
firing_rules {
 { X[i] -> v[i] : (i N) } =>
}
```


Note on Arrays and Structs

- Storage for arrays and structs is "sent" to receiving node. Let M be an array.

```
routing_rules {
  TRUE -> M1 <- M; M2 <-M;  // Error!
}
```

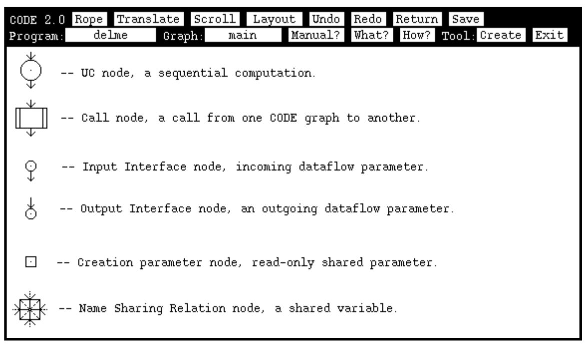
```
routing_rules {
  TRUE -> M1 <- M; M2 <-copy(M);  // Ok!
}
```

- "New" operator can allocate more storage, but better to get it back in a cycle.

```
M = new Mat2[5][N+7];
```

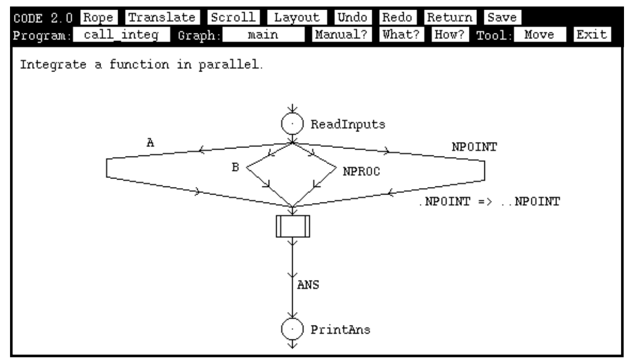
Other CODE Features

- Multiple graphs– like subroutines (recursion allowed).
  - Call nodes.
  - Interface nodes, Creation Parameter nodes.



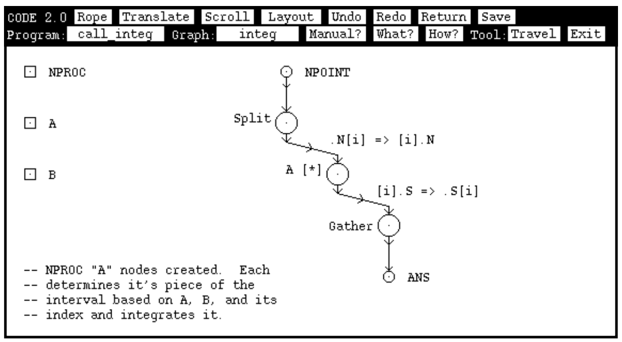
Graph Calling

- Revisit parallel integration.
- N-way parallelism using graph to do the work.
- Note extra dot in arc topology specification.



Graph Integ

- Creation parameters: read only.
  - "Broadcast" at graph creation time.
  - Instantiate graph for a particular problem size.
- Interface nodes connect dataflow arcs.



Spec of ReadInputs

output_ports { int NPROC; int NPOINT; real A; real B; }

vars { int nproc; int npoint; real a; real b; }

```
comp {
  nproc = ReadInt();
  a = ReadReal();
  b = ReadReal();
  npoint = ReadInt();
}

routing_rules {
  TRUE => NPOINT <- npoint; NPROC <- nproc; A <- a; B
   <- b;
}
```

Spec of Split

```
input_ports { int NPOINTS; }

output_ports { int N; }

vars { int NumProc; int NumPoint; }

firing_rules {
  NPOINTS -> NumPoint =>
}

comp {
  NumProc = NPROC;
}

routing_rules {
  { TRUE => N[i] <- (NumPoint/NumProc);
         NumPoint = NumPoint - (NumPoint/NumProc);
         NumProc = NumProc - 1;
            : (i NPROC)}
}
```


Spec of A

```
input_ports { int N; }

output_ports { real S; }

vars { int i; real a; real b; real val; int n; real h; }

firing_rules {
  N -> n =>
}

comp {
  i = NodeIndex[0];
  h = (B - A)/NPROC;
  val = simp(A + i*h, A + (i+1)*h, n);
}

routing_rules {
  TRUE => S <- val;
}
```

Spec of Gath

```
input_ports { real S; }
output_ports { real SUM; }

vars { real sum; int i; vect s[NPROC]; }

firing_rules {
  { S[i] -> s[i] : (i NPROC) } =>
}

comp {
  i = 0;
  sum = 0.0;
  while (i < NPROC) {
    sum = sum + s[i];
    i = i + 1;
  }
}

routing_rules {
  TRUE => SUM <- sum;
}
```


CODE Experimental Results

- **Portability**
  - **Sequent (shared memory) complete.**
  - **Serial (shared memory) complete.**
  - **PVM (message passing) begun [Vokkarne].**
  - **Ada (synchronous) complete in prototype.**
  - **No magic: programs can be biased towards an architecture.**
  - **CODE model easy for automatic translation.**
- **Ease of Use**
  - **Scientific evaluation difficult-- resource constraints.**
  - **Subjective evaluation by parallel programming class (4 students reported) generally positive.**
  - **Our evaluation: GUI needs work.**
- **Wide domain of applicability**
  - **Numerous smallish programs written.**
  - **Linear alg, particle simulation, differential equations, graph analysis...**

Results: Performance (Non-Expert)

- Casual results from parallel programming class (non-expert programmers).
  - Methodology: implement same algorithm in multiple parallel programming languages.
  - Performance relative to other languages.
    - Numeric comparisons senseless since different machines and language implementation qualities involved.
    - Success (good speedup) with other language implied success with CODE, 75% of the cases.
    - Success with CODE did not imply success with other languages .
  - Absolute performance: 75% achieved good speedups with CODE.
  - Model encouraged design for minimizing communications and maximizing locality.

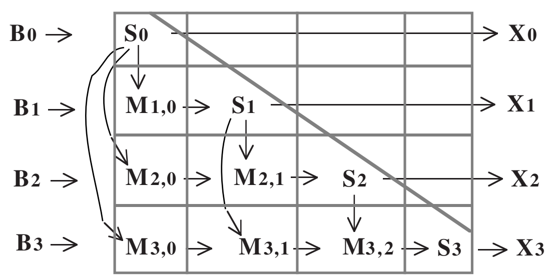
Results: Performance (Expert)

- Methodology : parallel computing Grad student prepares 3 programs.
  - Sequential– basis for speedup computation.
  - Parallel using CODE
  - Parallel using same target libraries as CODE (lower level).
- Compare speedups at problem sizes that show “bend” in speedup curve.
- Two Sequent Symmetry examples. Hand-parallel programs use FastThreads library.
  - Block triangular solver.
  - N-body particle simulation (Barnes-Hut).

Example: Block Triangular Solver

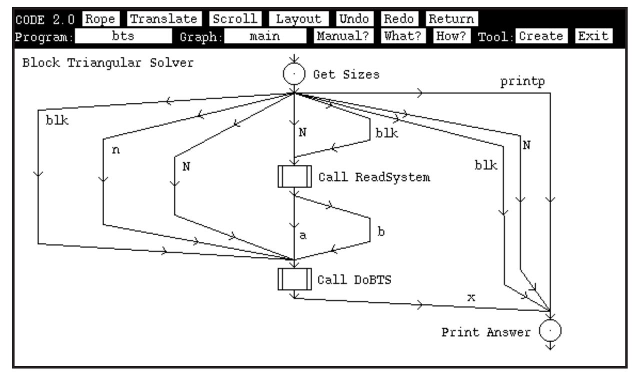
- Solve Ax = b for dense triangular A.

BTS - Data Flow



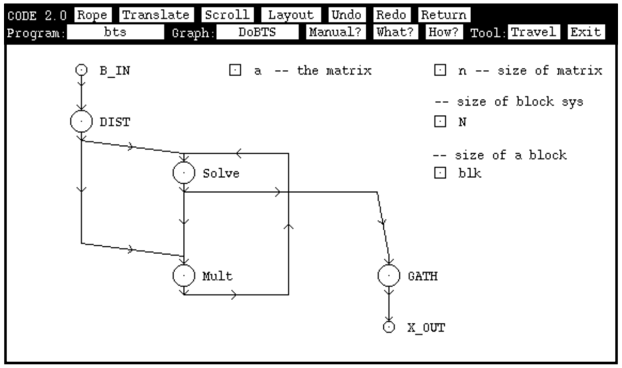
- Solve S sequentially.
- Multiply M by vector from above. Subtract from vector from left.
- Parallelism comes from M operations below an S.
- M and S operations-- standard sequential components.

Main Graph



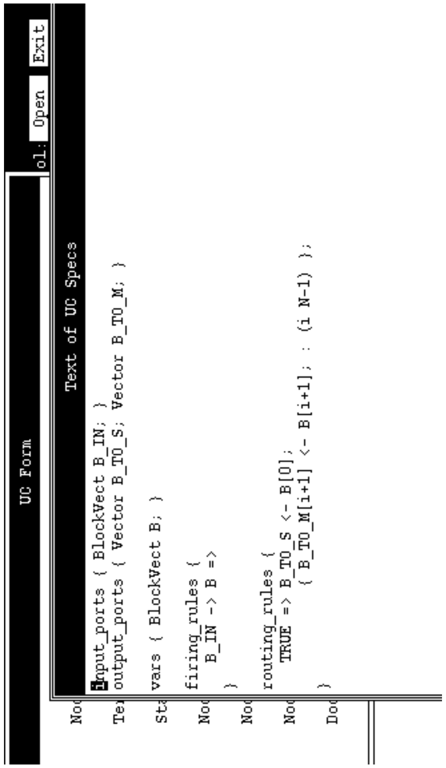
- Three graphs in program.
- Could use records to reduce number of arcs.

Graph DoBTS



- Solve sequentially solves (small) triangular systems.
- Node Mult is instantiated many times-- the source of the parallelism .
- DIST sends blocked vectors in from the left.
- GATH gathers them from the right.

Spec of Node DIST



Spec of Node Solve

Text of UC Specs

```
input_ports { Vector B_FROM_DIST; Vector B_FROM_M; }
output_ports { Vector B_TO_GATH; Vector B_TO_M; }
vars { Vector b; int WhichBlock; }
init_comp {
   WhichBlock = 0;
}

firing_rules {
   B_FROM_M -> b => ||
   B_FROM_DIST -> b =>
}

comp {
   solveblock(WhichBlock, a, b, blk);
   WhichBlock = WhichBlock + 1;
}

routing_rules {
   TRUE => B_TO_GATH <- b;
           { B_TO_M[i+WhichBlock] <- CopyVect(b, blk); : (i N-WhichBlock) };
}
```

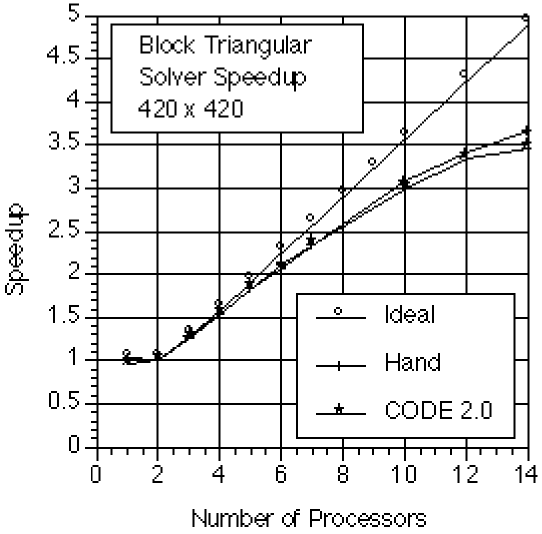

Spec of Node Mult

Text of UC Specs

```
input_ports { Vector B_FROM_S; Vector B_FROM_DIST; }
output_ports { Vector B_TO_S; }
vars { int WhichCol; Vector b_left; Vector b_up; }
init_comp {
   WhichCol = 0;
}

firing_rules {
   WhichCol > 0, B_FROM_S -> b_up => ||
   WhichCol == 0, B_FROM_DIST -> b_left, B_FROM_S -> b_up =>
}

comp {
   blkmult(NodeIndex[0], a, b_left, b_up, WhichCol, blk);
   WhichCol = WhichCol + 1;
}

routing_rules {
  WhichCol == NodeIndex[0] => B_TO_S <- b_left;
}
```

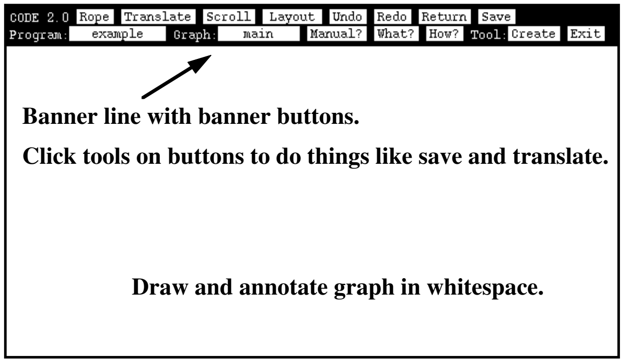
CODE 2.0 BTS Program Speedup

- **Relative to obvious sequential implementation.**
- **Run on 14 proc. Sequent Symmetry (FastThreads)**
- **Time measure by usec clock includes only system solving.**
- **System size 420 x 420.**



Using CODE 2

- **Run via "code2 progname.grf".**
- **Window appears. Draw and annotate graph within it.**



- **Mouse-- use <u>left</u> mouse button for everything.**

Cursor Tools

- Click cursor tools on "things" do perform actions.
  - Example: Click <u>Create</u> cursor on whitespace to draw a node. Pick the node type you want from the menu that appears.
  - Example: Click <u>Open</u> cursor on node, arc, Program button or Graph button to open attribute form.
- Click on Tool button to <u>change</u> current tool.
- Partial summary of tools:
  - Create: draw nodes and arcs.
  - Open: Open object attribute forms.
  - Travel: Move from one graph to another in same program.
  - Move: Move nodes, arcs, etc. within window.
  - Erase: Delete nodes, arcs, etc.

How To...

- Remember: use the left mouse button for everything.
- Exit CODE: Lick on Exit button.
- Save your work: Click on Save button (often!).
- Draw a node.

  Click Create cursor on whitespace. Pick the node type you want from the menu that appears
- Draw an arc.

  Click Create cursor on tip of arrow leaving node. Then click on end of arrow entering node. (Click on white space for arc intermediate points).
- Move a node: Click Move cursor on node.
- Open attribute form.

  Click Open cursor on Program button (contains program name), graph button (contains graph name), or a node or an arc. Attribute form will appear. Forms are hierarchical. They may contain sub-forms, menus, etc. Click on a field to fill it out.
- Move an open attribute form: drag its title bar.
- Close an attribute form.

  Click outside the form. Or click on another form.

More How To...

- Declare Start or Terminate Node: Set corresponding field in UC node attribute form.
- Enter or view type defintions: Open global types text field within Program attribute form. Example syntax:

```
type Vector is array of real;   // No sizes allowed!
type Matrix is array of Vector;
type Point is struct {
  real x;
  real y; }
```

- Enter or view function prototypes: Open global function signatures text field within Program attribute form. Example syntax:

```
void foo(int i; Matrix m;)
real VectSum(Vector v; int Size);
```

- Enter UC node spec: Open UC attribute form, field "Node Specification".
- Enter Name Sharing relation node spec: Open node's attribute form, field "NSRel Specification".

Still More How To...

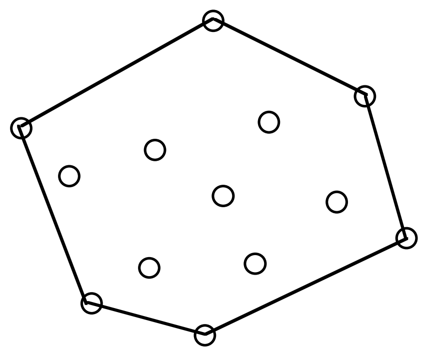
- Enter arc topology spec: open arc's attribute form, choose field "Arc Topology Rule." Example syntax:

```
.Y => .X          // one to one
.Y[i] => [i].X    // one to many
[i].Y => [i].X    // many to many
[i].Y => .X[i]    // many to one
```

- Translate graph: Click on Translate button. Choose "Serial" and look in shell window for error messages.
- Build and run program:
  - "cd progname.serial"
  - "make"
  - "progname"

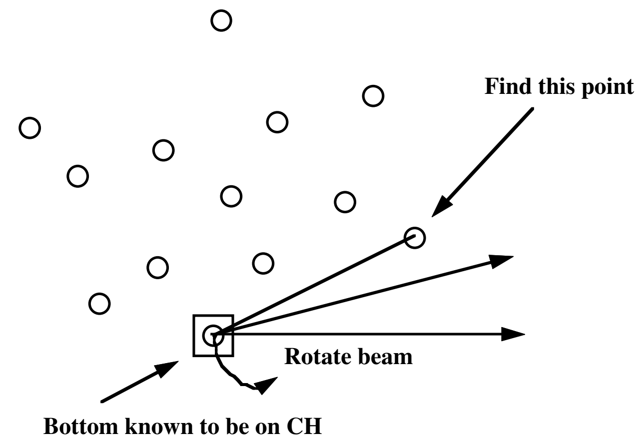
Exercise: Convex Hull Computation

- Find the "boundary" points in a set of points.



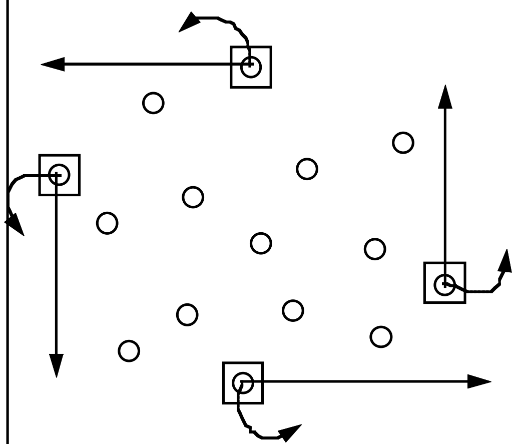
Sequential Algorithm

- Find an initial point on convex hull (top, bottom, left, or right).
- Rotate "laser beam" from point until next point found.
- Iterate.



Parallel Algorithm

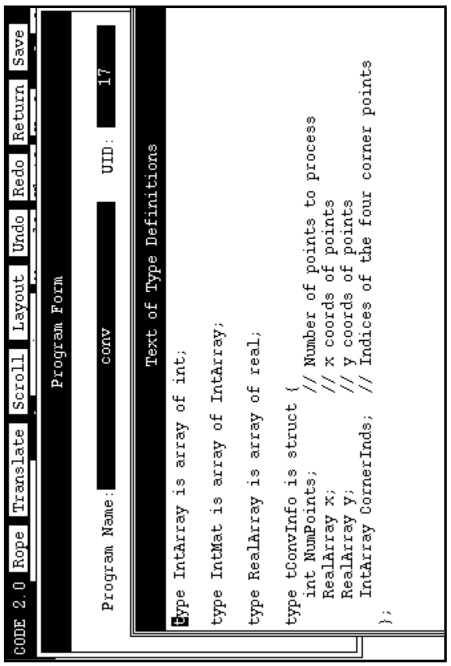
- Start from all four known points.
- Do each quadrant in parallel.



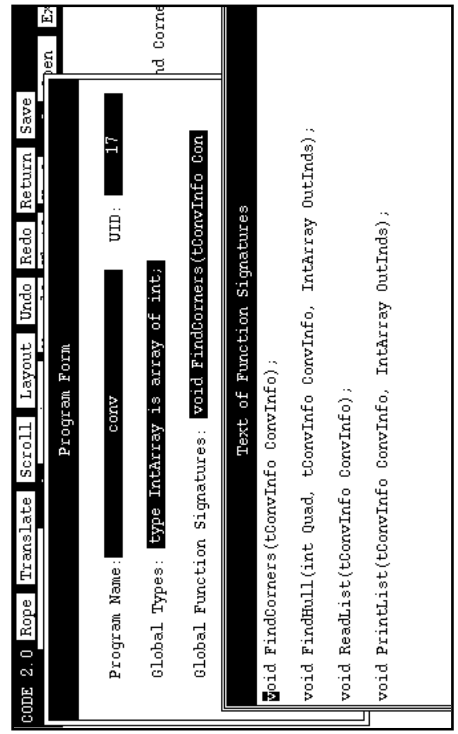
Sequential Building Blocks

- Some type defintions.
- Routine for reading list of points.
- Routine for finding the four corner points.
- Routine to find hull points within a quadrant.
- Routine to print hull.

Types



Functions



The Sequential Functions

- Already created (in convlib.c) and added to CODE program. Prototypes already entered.
- void ReadList(tConvInfo ConvInfo);
    - Reads points from file (like testdata) into ConvInfo.
    - Sets NumPoints; does not set CornerInds.
- void FindCorners(tConvInfo ConvInfo);
    - Computes indices of corner points (CornerInds).
    - Assumes Points & NumPoints already in ConvInfo.
- void FindHull(int Quad, tConvInfo ConvInfo, IntArray OutInds);
    - Computes indices of points in hull for given quadrant and stores them in OutInds.
    - Quad is one of 0 ... 3.
    - Assumes CornerInds computed and Points & NumPoints already in ConvInfo.

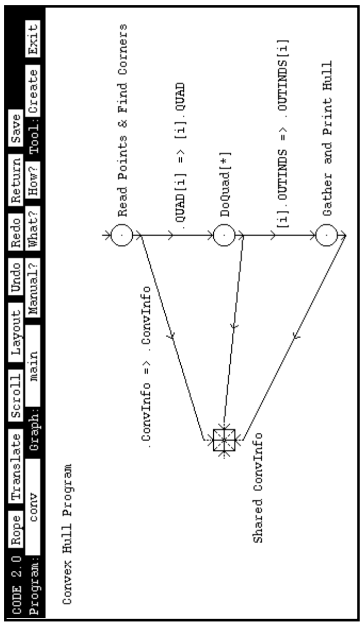
A Sequential Program

- Uses predefined sequential components.
- Assumes storage allocated for ConvInfo and OutInds.

```
ReadList(ConvInfo);
FindCorners(ConvInfo);

for (i = 0; i < 4; i++)
  FindHull(i, ConvInfo, OutInds[i]);   }

for (i = 0; i < 4; i++)
  PrintList(ConvInfo, OutInds[i]);
```

- For a parallel program, run the four calls in parallel with each other.

Convex Hull Program



Shared Data

- **Name Sharing Relation node purposes:**
  - **Declare shared variable ConvInfo.**
  - **Allocate storage for arrays x, y, and CornerInds within it.**

The node specification:

```
shared_vars {
  // alloc space for 40 points
  tConvInfo ConvInfo{x[40]; y[40]; CornerInds[4];};
}
```

Recall def of tConvInfo:

```
type tConvInfo  is struct {
  int NumPoints;
  RealArray x;
  RealArray y;
  IntArray CornerInds;
}
```

storage alloc.

Spec of Read Points...

```
output_ports { int QUAD; }
shared_vars { tConvInfo ConvInfo writer; }

comp {
  ReadList(ConvInfo);
  FindCorners(ConvInfo);
}

routing_rules {
  TRUE => { QUAD[i] <- i; : (i 4)};
}
```

Send Quad ID's to instances of DoQuad

Spec of DoQuad

```
input_ports { int QUAD; }
output_ports { IntArray OUTINDS; }
shared_vars { tConvInfo ConvInfo reader; }

vars { int Quad; IntArray OutInds[40]; }

firing_rules {
  QUAD -> Quad =>
}

comp {
  FindHull(Quad, ConvInfo, OutInds);
}

routing_rules {
  TRUE => OUTINDS <- OutInds;
}
```

Spec of Gather...

```
input_ports { IntArray OUTINDS; }
shared_vars { tConvInfo ConvInfo reader; }

vars { int i; IntMat IndMat[4]; }

firing_rules {  // recv points from the 4 quads.
  { OUTINDS[i] -> IndMat[i] : (i 4) } =>
}

comp {
  i = 0;
  while (i < 4) {  // print points in the 4 quads.
    PrintList(ConvInfo, IndMat[i]);
    i = i + 1;
  }
}
```


How to Obtain CODE 2

- **Anonymous ftp to pompadour.csres.utexas.edu [128.83.139.148].**
  - **Directory ParProg/code2.**
  - **See file INSTRUCTIONS for info on license agreement.**
  - **Postscript docs are there (no license needed).**
  - **These slides (Postscript) are there.**
- **Questions? Send email. Try Newton first.**
  - **newton@cs.utexas.edu**
  - **bwest@cs.utexas.edu**
  - **browne@cs.utexas.edu**