PARALLEL IMPLEMENTATION OF BEAMFORMER COEFFICIENT COMPUTATIONS

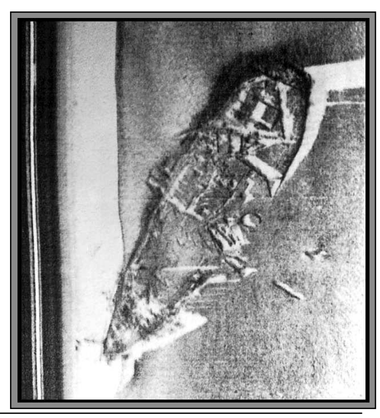
29 April 1999

CS 392C Methods and Techniques for Parallel Programming

Karen P. Watkins

PRESENTATION OVERVIEW

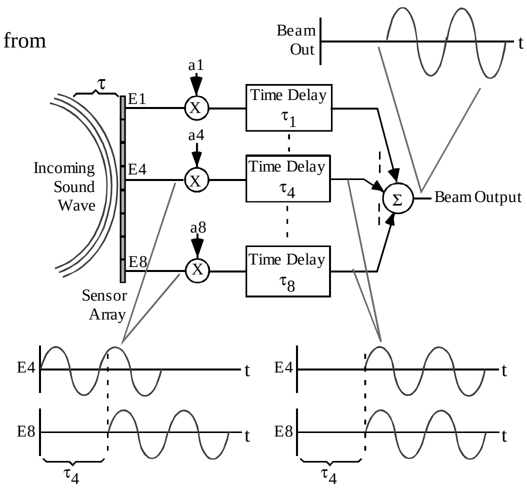
- Beamforming Background and Coefficient Computation
- Partitioning Coefficient Computation for Parallelism
- Serial Implementation
- Review of CODE 2 Graphical Programming
- CODE 2 Implementation
- MPI Implementation
- Results and Comparative Evaluation



Klien Assoc Inc.

TIME DELAY BEAMFORMING

- Spatial Filter Processes Signals from Distributed Sensors (Elements)
  - » Incoming sound wave arrives at each element at different times
  - » Elements convert the pressure wave into electronic signals
  - » Signals are appropriately time delayed to compensate for different arrival times
  - » Aligned signals are summed to produce increased signal amplitude from that arrival angle



a) Time of arrival at the sensors b) Signals after time delays imposed

MULTIPLE BEAM FORMATION

- Multiple Beams are Produced at Incrementally Different Angles
  - » Referred to as “Steering the Beam”
- Multiple Beams are then Displayed Side-by-Side to Form the Image

Sensor Array

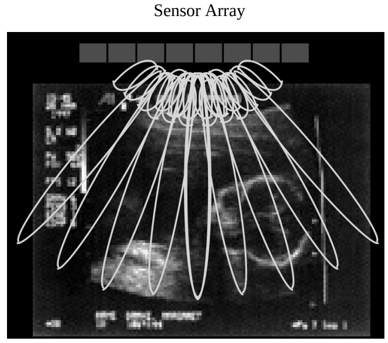
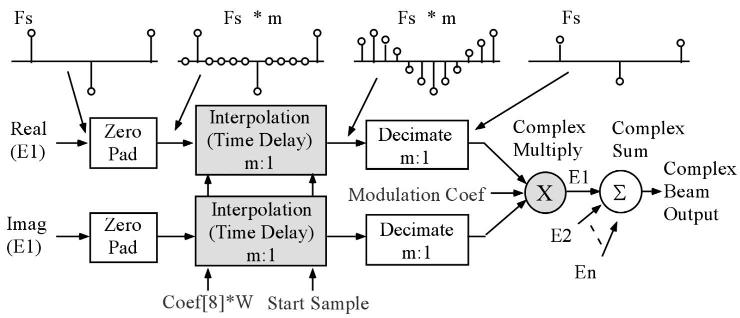


Image Produced using Multiple Beams

INTERPOLATION BEAMFORMING

- Technique Called Interpolation Beamforming Lowers the Required Sampling Rate to Accurately Steer the Beam
  - » Significantly reduces the number of processors and system throughput
- Coefficients are Applied to Each Element Signal in Interpolation Beamforming to Align the Signals More Efficiently

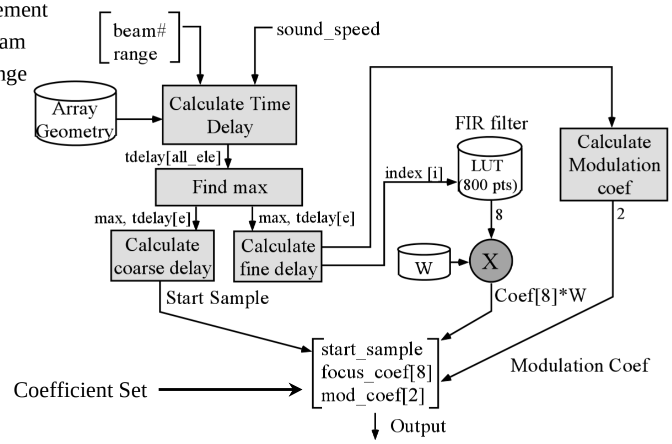


COEFFICIENT GENERATION

- Coefficients Are Computed Based Upon
  - » Each element's physical location
  - » Each desired beam pointing angle
  - » Each range increment
- To Achieve Highest Quality Images, Coefficients Are Dynamically Updated for
  - » Operating close to the array (focussing effect much like an optical camera)
  - » Localized changes in sound speed
- Leads to Massive Number of Coefficients to Compute in Real-time
  - » In the example presented here, 20 beams times 32 elements times 128 range increments times 11 coefficient values produce nearly 1 million coefficients

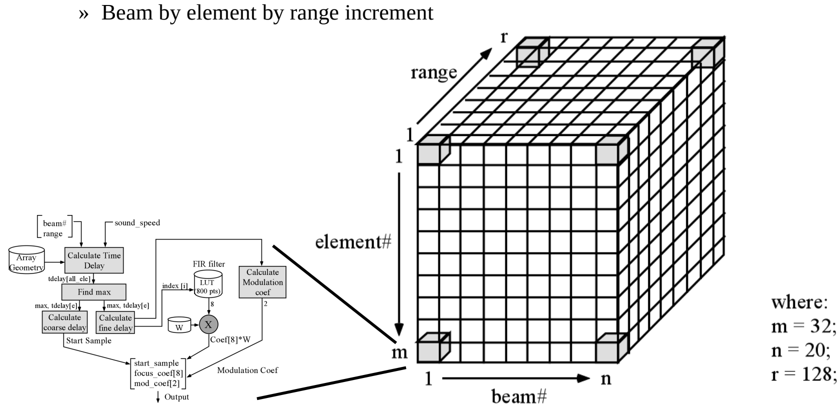
BEAMFORMER COEFFICIENT COMPUTATION ALGORITHM

- Produces One Set of Coefficients for
  - » One element
  - » One beam
  - » One range



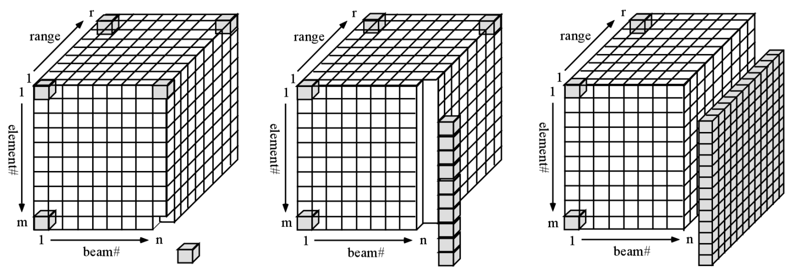
PARTITIONING OF COEFFICIENT COMPUTATIONS FOR PARALLELISM

- Complete Computation Can be Thought of as a 3-Dimensional Matrix
  - » Beam by element by range increment



DATA PARTITITONING ALTERNATIVES

- Many Ways to Exploit the Intrinsic Parallelism
- Alternatives Range from Fine Grain to Coarse Grain
- Parallel Systems Considered Here Favor Coarse Grain Solutions to Minimize Overhead



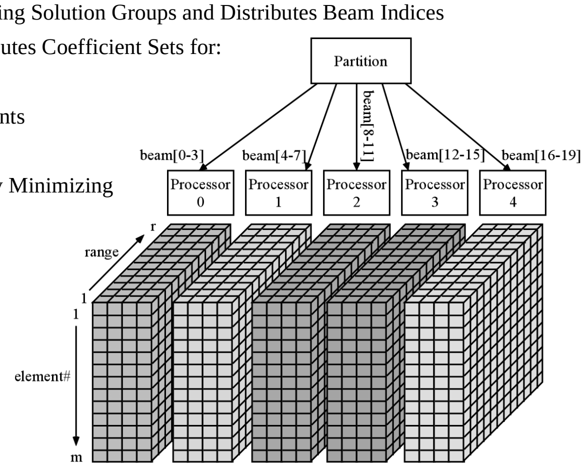
a) Each processor calculates one cell of the matrix

b) Each processor calculates one "pillar" of the matrix

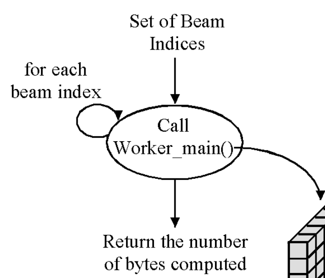
c) Each processor calculates one vertical plane of the matrix

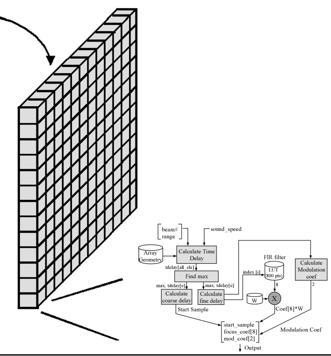
PREFERRED DATA PARTITIONING SOLUTION

- An Efficient Partitioning Solution Groups and Distributes Beam Indices
- Each Processor Computes Coefficient Sets for:
  - » Multiple beams
  - » All range increments
  - » All elements
- Reduces Overhead By Minimizing
  - » Communication
  - » Synchronization



CORE COMPUTATION STRUCTURE





- Core Computation Structure Used in
  - » Serial program
  - » CODE 2 program
  - » MPI program

SERIAL IMPLEMENTATION

- Serial Implementation Provides a Baseline for Comparison to Parallel Solutions

```
void main(argc,argv)
int argc;
char *argv[];
{
  sound_speed = (1.0/(12.0 * sound_speed));
  for (j=0; j<30; j++) {
    beam_index = 0;
    total_calc = 0;
    for (i=0; i<NUM_BEAMS; i++) {
      num_calc = worker_main(beam_index, sound_speed);
      beam_index++;
      total_calc += num_calc;
    }
  }
  return;
}
```

WORKER MAIN FUNCTION

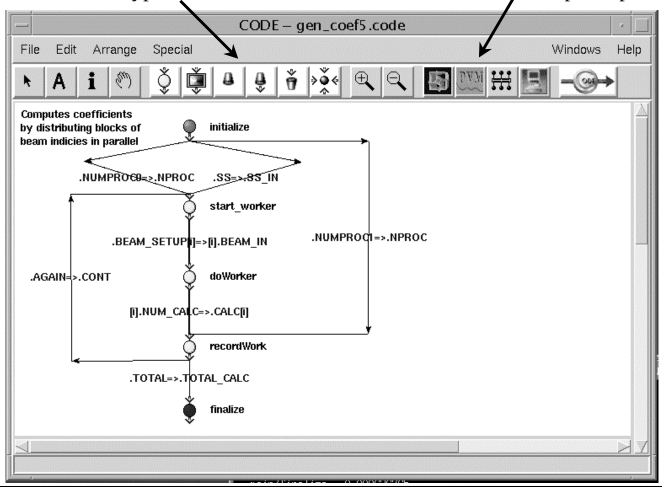
```
int worker_main(int beam_index, float sound_speed)
{
  if (isfirst_time == TRUE) {
    /* read array geometry, focussing points, filter coef LUT from files */
  }
  for (rng_i=0; rng_i<NUM_RNG; rng_i++) {
    range = focus_loc.foc_seg[rng_i].distance;
    steer = focus_loc.foc_seg[rng_i].steerdist[beam_index];

    /* time delay calculations */
    fine_time_delay = calc_time_delays(sound_speed, sensor, steer, range);

    /* coefficient calculations */
    calc_start_index(fine_time_delay, &rng_set[rng_i]);
    calc_focus_coef(fine_time_delay, weights_ptr, &rng_set[rng_i], coef_ptr);
    calc_modu_coef(fine_time_delay, &rng_set[rng_i]);
  }
  num_bytes_calc = NUM_RNG * sizeof(elem_set);
  return num_bytes_calc;
}
```


REVIEW OF CODE 2 GRAPHICAL PROGRAMMING SYSTEM

- User Specifies Parallelism Through Graph Topologies
- "Programming Language" Syntax is a Multigraph
  - » Nodes represent atomic computations and encapsulate
    - * sequential computation on local data
    - * firing rules that determine when computation can execute
    - * routing rules that specify the recipients of data produced by the computation
  - » Arcs represent unbounded FIFO queues that send data from one node to another (communication structure)
- Lazy Creation of Instances
  - » No element is instantiated at runtime until data first appears on its input port
- State Retention
  - » Once an element is created, it is retained and may be used many times
- Lazy Creation and State Retention Reduces Overhead
  - » Time and space is not wasted by creating all instances at start-up
  - » Time is saved by not destroying and then re-creating each element as needed

CODE 2 GRAPHICAL INTERFACE

Node Types Compile Options



FIRING RULES

- Firing Rules Serve Two Purposes
  - » Define the conditions under which a node is permitted to execute
  - » Describe the bindings that will occur
- Firing Rules are Very Flexible and Can Be Complicated
  - » Assignment statements can appear on the right side of the `=>` operator
  - » Rules for more than one arc input: `X -> v1, Y -> v2 =>`
  - » Node can have many firing rules
- CODE 2 Has Syntax to Replicate Guards, Bindings, or Entire Rules
  - » Syntax: { <thing> : (i N) } replicates <thing> for i taking on values from 0 to N-1
  - » Example: `{ I_SUM[i] -> p_sum[i] : (i N) } =>`
    - * node is allowed to fire when all ports have data
  - » Example: `{ I_SUM[i] -> p_sum[i] : (i N) => }`
    - * node is allowed to fire when any of the N ports have data

ROUTING RULES AND ARC ANNOTATIONS

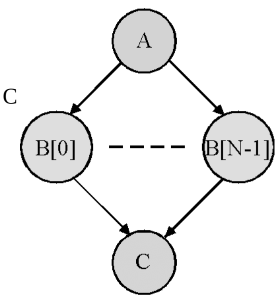
- Routing Rules Determine What Values Will Be Placed on Outgoing Arcs
  - » All Bindings are Performed from Left to Right
- Can Replicate Routing Rules to Place Data from an Array onto Multiple Output Ports
  - » Example: `TRUE => { OV[i] <- v[i]; : (i N) };`
- Arc Annotations Bind Port Names Between Nodes
  - » Allow graph structures to be dynamically configured
  - » Indices on the arc annotations indicate
    - * multiple ports on a node
    - * instantiation of nodes multiple times
    - * instantiation of graphs multiple times

CODE 2 TRANSLATION PROCESS

- CODE 2 Translation Process Provides Opportunities to Produce Efficient Code
- CODE 2 Implementers Can Write Translation Methods for Different Architectures and Apply Optimizations Techniques for these Architectures
- Translation Process Uses Runtime Libraries That Implement an Efficient Scheduler for CODE Nodes
  - » Exploits the atomic nature of nodes
  - » Uses non-preemptive scheduling
- Translation Methods Use Algorithms to Recognize the Most Common Constructs in Parallel Programming
  - » Example: Fan construct
  - » Apply optimizations to these constructs to produce efficient code

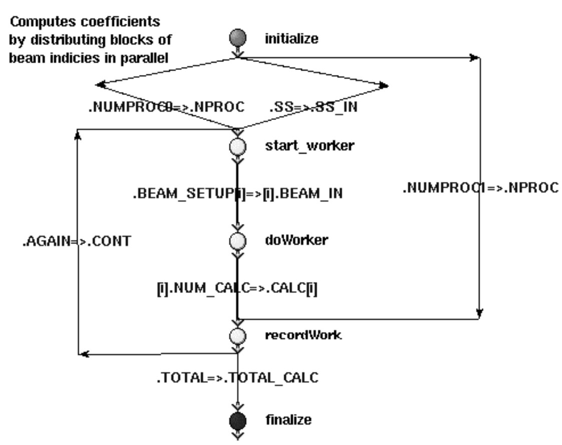
FAN PARALLEL PROGRAMMING STRUCTURE

- Fan Structure is Commonly Used in Parallel Computing
- Recognition Algorithms "Look for" These Structures
- Optimizations Applied to These Structures Include
  - » No queuing on the arcs
  - » No general firing rule checking for nodes B and C
  - » No general routing rule evaluation for B

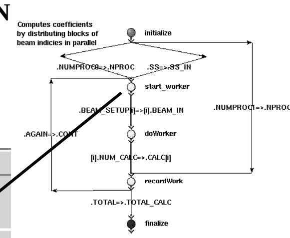


Computer Science *CS 392C Methods and Techniques for Parallel Programming*
The University of Texas at Austin

CODE 2 IMPLEMENTATION



START NODE DEFINITION



Firing Rules:

```
SS_IN -> ss, NPROC -> num_proc => ||
CONT -> i =>
```

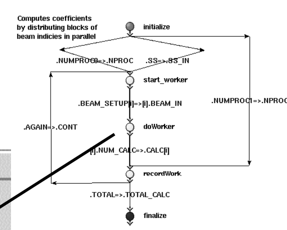
Routing Rules:

```
TRUE => { BEAM_SETUP[i] <- setup[i]; : (i num_proc) };
```

Every Computation:

```
setup = new worker_setup[num_proc];
i = 0;
stride = 20 / num_proc;

while (i<num_proc) {
  setup[i].start_beam = i*stride;
  setup[i].num_beam = stride;
  setup[i].ss = ss;
  i = i + 1;
}
```

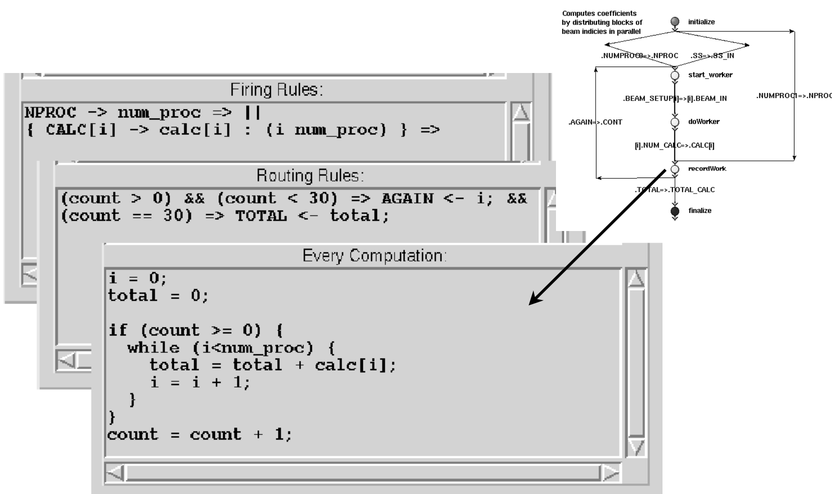

DO WORKER NODE DEFINITION



Every Computation:

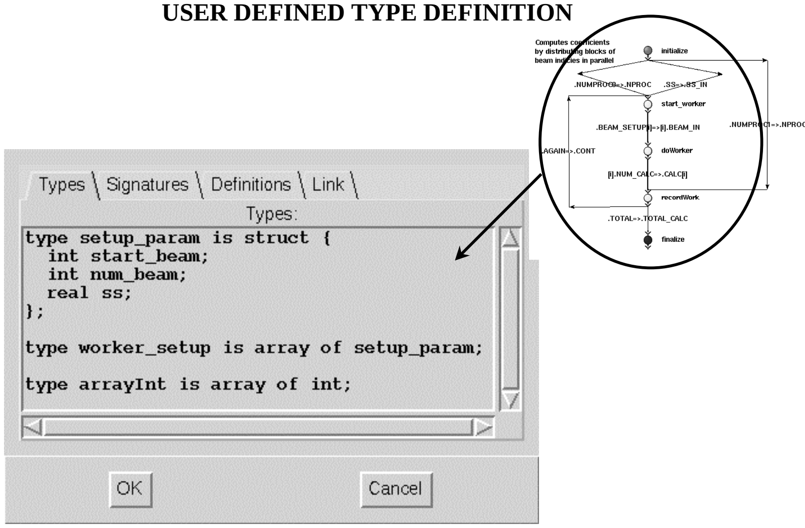
```
i = 0;
part_sum = 0;
stride = setup.num_beam;
beam = setup.start_beam;

while (i<stride) {
  num_calc = worker_main(beam, setup.ss);
  part_sum = part_sum + num_calc;
  beam = beam + 1;
  i = i + 1;
}
```

RECORD WORK NODE DEFINITION



Computer Science ***CS 392C Methods and Techniques for Parallel Programming***
The University of Texas at Austin

USER DEFINED TYPE DEFINITION

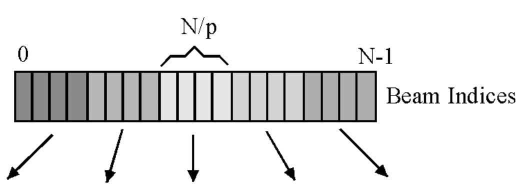


MPI METHODS APPLICABLE TO COEFFICIENT COMPUTATION

- MPI Provides Several Functions That "Naturally" Construct the Desired Topology
- MPI Broadcast
  - » Distributes common data to all processors in the communication group
- MPI Scatter
  - » One of many collective communication methods in MPI
  - » Divides array data into equal segments and distributes the segments among the processors
- MPI Barrier
  - » Synchronizes processes within a communication group
- MPI Send / Receive
  - » Blocking send and receive

SCATTERING OF BEAM INDICES

- MPI Scatter Divides Vector of Beam Indices into Equal Segments Based on the Number of Processors
- One Group of Beam Indices is Distributed to Each Processor to Compute Coefficients



MPI IMPLEMENTATION

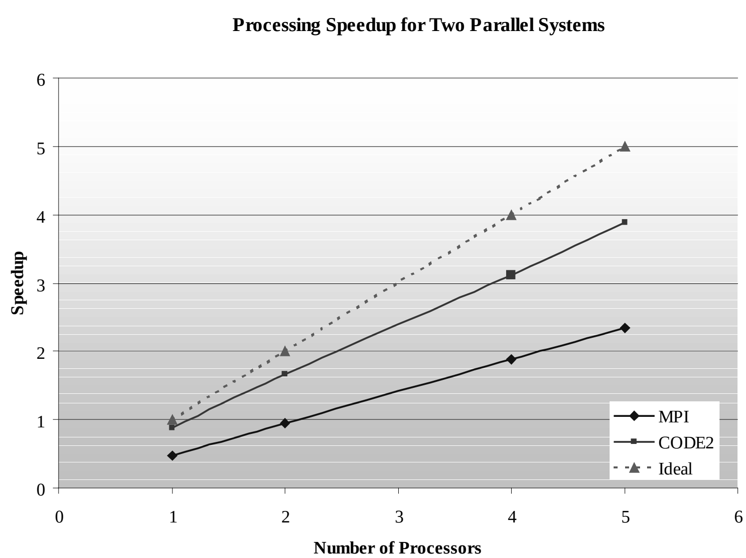
```
void main(argc,argv)
{
  stride = NUM_BEAMS/nproc;
  for (j=0; j<30; j++) {
    MPI_Barrier(MPI_COMM_WORLD);
    total_calc = 0, part_sum = 0, num_rcvd = 0;
    for (i=0; i<NUM_BEAMS; i++) {
      send_buf[i] = i;
    }

    sound_speed = (1.0/(12.0 * sound_speed));
    MPI_Bcast(&sound_speed, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

    /* construct packet and start all workers */
    MPI_Scatter(send_buf, stride, MPI_INT, recv_buf,
                          stride, MPI_INT, root, MPI_COMM_WORLD);

    for (i=0; i<stride; i++) {
      beam_index = recv_buf[i];
      num_calc = worker_main(beam_index, sound_speed);
      part_sum += num_calc;
    }
```


MPI IMPLEMENTATION (Continued)

```
    if (my_rank == 0) { /* master definition */
      total_calc += part_sum;

      /* get any data from workers */
      while (num_rcvd < (nproc - 1)) {
        MPI_Recv(&work_sum, 1, MPI_INT, MPI_ANY_SOURCE,
                                   DATA_MSG, MPI_COMM_WORLD, &status);
        total_calc += work_sum;
        num_rcvd++;
      }
    } else {   /* worker definition */
      MPI_Send(&part_sum, 1, MPI_INT, root, DATA_MSG, MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
  return;
}
```

PERFORMANCE TESTING AND EVALUATION

- Serial Implementation Execution Time Was Measured as a Performance Metric
- CODE 2 and MPI Execution Times Were Recorded
  - » 1, 2, 4, and 5 processors
- Number of Processors Were Chosen to
  - » Maintain load balance
- Measurements Made Sun E5000, 8 Processor, Solaris 2.6 (enzo)



TECHNICAL RESULTS

- Both CODE 2 and MPI Demonstrated Linear Speedups
- CODE 2 Outperformed MPI with Respect to Speedup and Execution Time
- CODE 2 Produced More Efficient Solution By Applying Optimization Techniques
  - » Recognition of fan structure avoided queuing on arcs, firing and routing rule checks
  - » Architecture specific optimizations (SMP)
    - * e.g. runtime library uses light weight threads
  - » Non-preemptive scheduler that exploits the atomic nature of nodes
  - » Lazy creation and state retention
- MPI Inherently Has Overhead That Reduces Throughput Performance
  - » e.g. multiple copy operations to pass data from one process to another

COMPARATIVE EVALUATION

CODE 2

- Graphical Interface is Easy to Use
- Visual Representation of the Structure Helped to Identify Parallelism and Granularity
- Little Effort Expended to “Optimize” the Design
  - » Optimization is “built-in” to CODE
- Automatically and Dynamically Schedules Event Execution
- Supports Various Standard Parallel Message Passing Protocols
  - » MPI, PVM, SMP

MPI

- Built-in Communication Structures Produce Code
  - » Intuitive, easy to read, and (can be) compact
  - » Straight-forward design
- Any “Optimization” is Under the Control of the User
- No Schedulers