Tutorial on Parallel Programming with Linda
## Linda Tutorial Outline

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linda Basics</td>
<td>3</td>
</tr>
<tr>
<td>Linda Programming Environment</td>
<td>21</td>
</tr>
<tr>
<td><em>Exercise #1 Hello World</em></td>
<td>26</td>
</tr>
<tr>
<td><em>Exercise #2 Ping/Pong</em></td>
<td>33</td>
</tr>
<tr>
<td>Parallel Programming with Linda</td>
<td>34</td>
</tr>
<tr>
<td><em>Exercise #3 Integral PI</em></td>
<td>52</td>
</tr>
<tr>
<td>Linda Implementation</td>
<td>54</td>
</tr>
<tr>
<td>Linda versus Competition</td>
<td>68</td>
</tr>
<tr>
<td><em>Exercise #4 Monte Carlo PI</em></td>
<td>76</td>
</tr>
<tr>
<td><em>Exercise #5 Matrix Multiplication</em></td>
<td>78</td>
</tr>
</tbody>
</table>
Linda Basics
Linda Technology Overview

- Six simple commands enabling existing programs to run in parallel
- Complements any standard programming language to build upon user’s investments in software and training
- Yields portable programs which run in parallel on different platforms, even across networks of machine from different vendors
What’s in Tuple Space

- A Tuple is a sequence of typed fields:

  (“Linda is powerful”, 2, 32.5, 62)

  (1,2, “Linda is efficient”, a:20)

  (“Linda is easy to learn”, i, f(i))
Tuple Space provides

- Process creation
- Synchronization
- Data communication

These capabilities are provided in a way that is logically independent of language or machine.
Operations on the tuple space

- **Generation**
  - `eval`
  - `out`

- **Extraction**
  - `in`
  - `inp`
  - `rd`
  - `rdp`
Linda Operations: Generation

- **out**
  - Converts its arguments into a tuple
  - All fields evaluated by the outing process

- **eval**
  - Spawns a “live tuple” that evolves into a normal tuple
  - Each field is evaluated separately
  - When all fields are evaluated, a tuple is generated
Linda Operations: Extraction

- **in**
  - Defines a template for matching against Tuple Space
  - Either finds and removes matching tuple or blocks

- **rd**
  - Same as in but doesn’t remove tuple

- **inp, rdp**
  - Same as in and rd, but returns false instead of blocking
Out/Eval

- Out evaluates its arguments and creates a tuple:
  
  \[
  \text{out("cube", 4, 64);}
  \]

- Eval does the same in a new process:
  
  \[
  \text{eval("cube", 4, f(i));}
  \]
In/Rd

- These operations would match the tuples created by the `out` and `eval`.

\[
\text{In(“cube”, 4, ?j);}
\]
\[
\text{rd(“cube”, 4, ?j);}
\]

*As a side effect, \( j \) would be set to 64*
Using the Virtual Shared Memory

A: `out('south', imin, jmin, u(2:nxloc+1,2))`

A2: `...`

B: `do i = 1, 3
eval('table entry', i, f(i))
end do`

C: `imin = 17
jmax = 32
in('south', imin, jmax+1, ?u(2:nxloc+1,nyloc+2))`

D: `i = 3
rd('table entry', i, ?fi)`
Tuple/Template matching rules

- Same number of fields in tuple and template
- Corresponding field types match
- Fields containing data must match
C Tuple data types

- In C, tuple fields may be of any of the following types:
  - int, long, short, and char, optionally preceded by unsigned.
  - float and double
  - struct
  - union
  - arrays of the above types of arbitrary dimension
  - pointers must always be dereferenced in tuples.
Fortran Tuple types

- In Fortran, tuple fields may be of these types
  - Integer (*1 through *8), Real, Double Precision,
  - Logical (*1 through *8), Character, Complex, Complex*16
  - Synonyms for these standards types (for example, Real*8).
  - Arrays of these types of arbitrary dimension, including
    multidimensional arrays, and/or portions thereof.
  - Named common blocks
Array fields

- The format for an array field is *name:*len

```c
char a[20];
out("a", a:); all 20 elements
out("a", a:10); first 10 elements
in("a", ?a:len); stores # recvd in len
```
Matching Semantics

- Templates matching no tuples will block (except inp/rdp)
- Templates matching multiple tuples will match non-deterministically
- Neither Tuples nor Templates match oldest first
- These semantics lead to clear algorithms without timing dependencies!
Linda Distributed Data Structures

- Linda can be used to build distributed data structures in Tuplespace
- Easier to think about than “data passing”
- Atomicity of Tuple Operations provides data structure locking
Linda Distributed Data Structures (examples)

- **Counter**
  
  ```
  in("counter", "name", ?cnt);
  out("counter", "name", cnt+1);
  ```

- **Table**
  
  ```
  for(i=0; i<n; i++)
    out("table", "name", elem[i]);
  ```
Linda Distributed Data Structures (examples)

- **Queue**

```plaintext
init()
{
    out("head", 0);
    out("tail", 0);
}

put(elem)
{
    in("tail", ?tail);
    out("elem", tail, elem);
    out("tail", tail + 1);
}

take(elem) {
    in("head", ?head);
    out("elem", head, elem);
    out("head", head + 1);
}
```
The Linda Programming Environment
Software Development with Linda

- Step 1: Develop and debug sequential modules
- Step 2: Use Linda Code Development System and TupleScope to develop parallel version
- Step 3: Use parallel Linda system to test and tune parallel version
Linda Code Development System

- Implements full Linda system on a single workstation
- Provides comfortable development environment
- Runs using multitasking, permitting realistic testing
- Compatible with TupleScope visual debugger
Parallel “Hello World”

```c
#define NUM_PROCS 4
real_main()
{
    int i, hello_world();
    out("count", 0);
    for (i=1; i<=NUM_PROCS; i++)
    {
        eval("worker", hello_world(i));
        in("count", NUM_PROCS);
        printf("all processes done.\n");
    }
}

hello_world(i)
int i;
{
    int j;
    in("count", ?j);
    out("count", j+1);
    printf("hello world from process %d, count %d\n", i, j);
}
```
Using Linda Code Development System

% setenv LINDA_CLC cds
% clc -o hello hello.cl
CLC (V3.1 CDS version)
hello.cl:10: warning --- no matching Linda op.
% hello
Linda initialization complete.
Hello world from process 3 count 0
Hello world from process 2 count 1
Hello world from process 4 count 2
Hello world from process 1 count 3
All processes done.
all tasks are complete (5 tasks).
Hands on Session #1 - Hello World

- Compile hello.cl using the Code Development System.
- Run the program.
Linda Termination

Linda Programs terminate in three ways:

- Normal termination: when all processes (real_main and any evals) have terminated, by returning or calling lexit()

- Abnormal termination: any process ends abnormally

- lhalt() termination: any process may call lhalt()

The system will clean up all processes upon termination.

*Do not call exit from within Linda programs!*
TupleScope Visual Debugger

- X windows visualization and debugging tool for parallel programs
- Displays tuple classes, process interaction, program code, and tuple data
- Contains usual debugger features like single-step of Linda operation
- Integrated with source debuggers such as dbx, gdb.
Linda TupleScope

```
void process()
{
    rd("start");
    for (i = 0; i < loops; ++i) {
        /* hit the ball */
        out("ping");
        /* receive the ball */
        in("pong");
    }
    return(0);
}
```
Debugging with the Linda Code Development System

Compile program for TupleScope and dbx:

```
clc -g -o hello -linda tuple_scope hello.cl
```

Run program, single stepping until desired process is evaluated
Middle click on process icon
Set breakpoint in native debugger window
Turn off TupleScope single stepping
Control process via native debugger
TCP Linda

- TCP Linda programs are started via `ntsnet` utility

- Ntsnet will:
  - Read `tsnet.config` configuration file
  - Determine network status
  - Schedule nodes for execution
  - Translate directory paths
  - Copy executables
  - Start Linda process on selected remote machines
  - Monitor execution
  - Etc, etc.
Running a program with `ntsnet`

- Create file `~/.tsnet.config` containing the machine names:
  ```
  Tsnet.Appl.nodelist: io ganymede rhea electra
  ```

- Compile the program with TCP Linda
  ```
  % setenv LINDA_CLC linda_tcp
  % clc -o hello hello.cl
  ```

- Run program with `ntsnet`
  ```
  % ntsnet hello
  ```
Hands on Session #2 - Ping Pong

- This exercise demonstrates basic Linda operations and TupleScope

- Write a program that creates two workers called ping() and pong(). Ping() loops, sending a “ping” tuple and receiving a “pong”, while pong() does the opposite.

- Ping and pong should agree on the length of the game, and terminate when finished.

- Compile and run the program using Code Development System and TupleScope.
Parallel Programming with Linda
Parallel Processing Vocabulary

- Granularity: ratio of computation to communication
- Efficiency: how effectively the resources are used
- Speedup: performance increase as CPU’s are added
- Load Balance: is work evenly distributed?
Linda Algorithms

- Live Task
- Master/Worker
- Domain Decomposition
- Owner Computes

*This is not an exhaustive list: Linda is general and can support most styles of algorithms*
Live Task Algorithms

- Simplest Linda Algorithm
- Master evals task tuples
- Retrieves completed task tuples

Caveats:
  - simple parameters only
  - watch out for scheduling problems
  - think about granularity!
Live Task Algorithms

Sequential Code

```c
main()
{
    /* initialize a[], b[] */
    for (i=0; i< LIMIT; i++)
        res[i]=comp(a[i], b[i]);
}
```

Linda Code

```c
real_main()
{
    /* initialize a[], b[] */
    for (i=0; i< LIMIT; i++)
        eval("task", i, comp(a[i],b[i]));

    for (i=0; i< LIMIT; i++)
        in("task", i, ?res[i]);
}
```
Master/Worker Algorithms

- **Separate Processes and Tasks**
  Tasks become lightweight

- **Master**
  - evals workers
  - generates task tuples
  - consumes result tuples

- **Worker**
  - loops continuously
  - consumes a task
  - generates a result
Dynamic Load Balancing

- Task to do
- Task in progress
- Completed task
Master/Worker Algorithms: sequential code

```c
main()
{
    RESULT r;
    TASK t;

    while (get_task(&t)) {
        r = compute_task(t);
        update_result(r);
    }
    output_result();
}
```
Master/Worker Algorithms: parallel code

```c
real_main()
{
    int i;
    RESULT r;
    TASK t;

    for (i=0; i<NWORKER; i++)
        eval("worker", worker());
    for (i=0; get_task(&t); i++)
        out("task", t);
    for (; i; --i)
        in("result", ?r);
    update_result(r);
}

worker()
{
    RESULT r;
    TASK t;

    while (1) {
        in("task", ?t);
        r = compute_task(t);
        out("result", r);
    }
}
```
Master/Worker Algorithms:

- To be most effective, you need:
  - Relatively independent tasks
  - More tasks than workers
  - May need to order tasks

- Benefits:
  - Easy to code
  - Near ideal speedups
  - Automatic load balancing
  - Lightweight tasks
Domain Decomposition Algorithms

- Specific number of processes in a fixed organization
- Relatively fixed, message/passing style of communication
- Processes work in lockstep, often time steps
A PDE Example

\[ \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \]

\[ u(x,y,t+dt) = u(x,y,t) + \frac{dt}{dx^2} \{ u(x+dx,y,t) + u(x-dx,y,t) - 2u(x,y,t) \} + \frac{dt}{dy^2} \{ u(x,y+dy,t) + u(x,y-dy,t) - 2u(x,y,t) \} \]
subroutine real_main
common /parms/ cs, cy, nts

... GET INITIAL DATA ...

out('parms common', /parms/)
np = 0
do ix = 1, nx, nxloc
   ixmax = min(ix+nxloc-1, nx)
   do iy = 1, ny, nyloc
      iymax = min(iy+nyloc-1, ny)
      np = np + 1
      if (ix.gt.nxloc .or. iy.gt.nyloc) then
         eval('worker', worker(ix, ixmax, iy, iymax))
      endif
      out('initial data', ix, iy, u(ix:ixmax, iy:iymax))
   enddo
enddo
call worker(1, min(nxloc,nx), 1, min(nyloc,ny))
do i = 1, np
   in('result', ixmin, iymin, ?u(ixmin:ixmax, iymin:iymax))
endo
d:return
ed
Worker Routines - I

subroutine worker(ixmin, ixmax, iymin, iymax)
common /parms/ cx, cy, nts
dimension uloc(NXLOCAL+2, NYLOCAL+2, 2)

nxloc = ixmax - ixmin + 1
nyloc = iymax - iymin + 1

rd('parms common', ?/parms/)
in('initial data', ixmin, iymin, ?uloc(2:nxloc+1,2:nyloc+1))

iz = 1
do it = 1, nts
    call step(ixmin, ixmax, iymin, iymax, NXLOCAL+2,
*             nxloc, nyloc, iz, uloc(1,1,iz), uloc(1,1,3-iz))
    iz = 3 - iz
enddo

out('result id', ixmin, ixmax, iymin, iymax)
out('result', ixmin, iymin, uloc(2:nxloc+1, 2:nyloc+1, iz))

return
end
subroutine step(ixmin, ixmax, iymin, iymax, nrows, *
    nxloc, nyloc, iz, u1, u2)
common /parms/ cx, cy, nts
dimension u1(nrows, *), u2(nrows, *)

if (ixmin.ne.1) out('west', ixmin, iymin, u1(2, 2:nyloc+1))
if (ixmax.ne.nx) out('east', ixmax, iymin, u1(nxloc+1, 2:nyloc+1))
if (iymax.ne.ny) out('north', ixmin, iymax, u1(2:nxloc+1, nyloc+1))
if (iymin.ne.1) out('south', ixmin, iymax, u1(2:nxloc+1, 2))

if (ixmin.ne.1) in('east', ixmin-1, iymin, ?u1(1,2:nyloc+1))
if (ixmax.ne.nx) in('west', ixmax+1, iymin, ?u1(nxloc+2, 2:nyloc+1))
if (iymin.ne.1) in('north', ixmin, iymin-1, ?u1(2:nxloc+1, 1))
if (iymax.ne.ny) in('south', ixmin, iymax+1, ?u1(2:nxloc+1, nyloc+2))

    do ix = 2, nxloc + 1
        do iy = 2, nyloc + 2
            u2(ix,iy) = u1(ix,iy) +
            cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
            cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
        enddo
    enddo
return
end
Owner-Computes Algorithm

- Share loop iterations among different processors
- Iteration allocation can be dynamic
- Allows for parallelization with little change to the code
- Good for parallelizing complex existing sequential codes
Owner-Computes: sequential code

```c
main()
{
    /* lots of complex initialization */
    for (ol=0; ol<loops; ++ol) {
        for (i=0; i<n; ++i) {
            ...
            elem[i] = f(...); /* complex calculation */
            ...
        }
    }
    /* more complex calculations using elem[]; */
}
/* output results to disk...*/
```
Parallel Owner Computes

real_main() {
    out("task", 1);
    for (i=1; i<NUMPROCS; ++i)
        eval(old_main(i));
    old_main(0);
}

old_main(id)
int id;
{
    /* complex init */
    for (ol=0; ol<loops; ++ol) {
        for (i=0; i<n; ++i) {
            if (!check()) continue;
            ...
            elem[i] = f(...);
            ...
            log_data(i, elem[i]);
        }
        gatherscatter();
        /* more complex calc */
    }
    /* output results to disk...*/
}

check(){
    static int next=0, count=0;
    if (next==0) {
        in("task", next);
        out("task", next+1);
    }
    if (++count == next) {
        next=0; return 1;
    }
    return 0;
}

log_data(i, val){
    local_results[i].id = i;
    local_results[i].val = val;
}

gatherscatter(){
    if (myid==0)
        /* master ins local results */
    ...
    else
        /* worker ins local results */
    ...
}
Hands on Session #3 - Live Task PI

- Convert sequential C program to parallel using live task method.
- $\pi$ is the integral of $4/(1+x^2)$ from 0 to 1.
main(argc, argv)
int argc;
char *argv[];
{
    nsteps = atoi(argv[1]);
    step   = 1.0/(double)nsteps;
    pi=subrange(nsteps,0.0,step)*step;
}

double subrange(nsteps,x,step)
{
    double result = 0.0;
    while(nsteps>0) {
        result += 4.0/(1.0+x*x);
        x += step;
        nsteps--;
    }
    return(result);
}
Linda Implementation
Linda Implementation Efficiency

- Tuple usage analysis and optimization is the key to Scientific’s Linda systems
- Optimization occurs at compile, link, and runtime

In general

- Satisfying an in requires touching fewer than two tuples
- On distributed memory architectures, communication pattern optimizes to near message-passing
Implementation

- 3 major components:
  
  Compile Time: Language Front End
  
  * supports Linda syntax
  * supports debugging
  * supports tuple usage analysis

  Link Time: Tuple-usage Analyzer
  
  * optimizes run-time tuple storage and handling

  Run Time: Linda Support Library
  
  * initializes system
  * manages resources
  * provides custom tuple handlers
  * dynamically reconfigures to optimize handling
Linda Compile-time Processing

Linda Source Code

Parser

Linda Engine

Native Source Code

Native Compiler

Native Object Code

Tuple Usage Data

Linda Object File
Linda Link-time Processing

- Linda Object Files
  - Analyzer
    - Generated C Source Code
    - Native Compiler
      - Generated C Object Code
    - Object Code
      - Loader
        - Executable
  - Other Object Files
  - Linda Library
Tuple Usage Analysis

- Converts Linda operations into more efficient, low level operations
- Phase I partitions Linda operations into disjoint sets based on tuple and template content

Example:
- `out ("date", i, j)`
  - *can never match*
- `in("sem", ?i)`
Tuple Usage Analysis

- Phase II analyzes each partition found in Phase I
- Detects patterns of tuple and template usage
- Maps each pattern to a conventional data structure
- Chooses a runtime support routine for each operation
Linda Compile-Time Analysis Example

/* Add to order task list */
out("task", ++j, task_info) /* S1 */

/* Extend table of squares */
out("squares", i, i*i) /* S2 */

/* Consult table */
rd("squares", i, ?i2) /* S3 */

/* Grab task */
in("task", ?t, ?ti) /* S4 */
Linda Compile-Time Analysis Example

- Phase I: two partitions are generated:
  \[ P_1 = \{ S_2, S_3 \} \quad P_2 = \{ S_1, S_4 \} \]

- Phase II: each partition optimized:
  \[ P_1: \]
  - Field 1 can be suppressed
  - Field 3 is copy only (no matching)
  - Field 2 must be matched, but key available \( \Rightarrow \) hash implementation

  \[ P_2: \]
  - Field 1 can be suppressed
  - Fields 2 & 3 are copy only (no matching) \( \Rightarrow \) queue implementation
Linda Compile-Time Analysis

- Associative matching reduced to simple data structure lookups

- Common set paradigms are:
  - counting semaphores
  - queues
  - hash tables
  - trees

- In practice, exhaustive searching is never needed
Run Time Library

- Contains implementations of set paradigms for tuple storage
- Structures the tuple space for efficiency
- Families of implementations for architecture classes
  - Shared-memory
  - Distributed-memory
  - Put/get memory
TCP Linda runtime optimizations

- **Tuple rehashing**
  Runtime system observes patterns of usage, remaps tuples to better locations
  Example: Domain decomposition
  Example: Result tuples

- **Long field handling**
  Large data fields can be stored on outing machine
  We know they are not needed for matching
  Bulk data transferred only once
Hints for Aiding the Analysis

Use String Tags

- `out("array elem", i, a[i])`
- `out("task", t);`

- Code is self documenting, more readable
- Helps with set partitioning
- *No runtime cost!*

Hints for Aiding the Analysis

Use care with hash keys

- Hash key is non-constant, always actual
  
  - out("array elem", iter, i, a[i])
  - in("array elem", iter, i, ?val)

- Analyzer combines all such fields (fields 2 and 3)

- Avoid unnecessary use of formal in hash field (common in cleanup code)
  
Linda vs. the Competition
Portable Parallel Programming

Four technology classes for Portable Parallel Programming:

- Message Passing - *the machine language of parallel computing*
- Language extensions - *incremental build on traditional languages*
- Inherently Parallel Languages - *elegant but steep learning curve*
- Compiler Tools - *the solution to the dusty deck problem?*
Portable Parallel Programming: the major players

Four technology classes for Portable Parallel Programming:

- Message Passing - *MPI, PVM, Java RMI*...
- Language extensions - *Linda, Java Spaces*...
- Inherently Parallel Languages - ??
- Compiler Tools - *HPF*
Why not message passing?

- Message passing is the machine language of distributed-memory parallelism
  
  *It’s part of the problem, not the solution*

- Linda’s Mission:
  
  *Comparable efficiency with much greater ease of use*
Linda is a high-level approach

- Point-to-point communication is trivial in Linda, so you can do message passing if you must...

- ... but Linda’s shared associate object memory is extremely hard to implement in message passing

- Message Passing is a low-level approach
/* Receive data from master */
msgtype = 0;
pvm_recv(-1, msgtype);
pvm_upkint(&nproc, 1, 1);
pvm_upkint(tids, nproc, 1);
pvm_upkint(&n, 1, 1);
pvm_upkfloat(data, n, 1);
/* Do calculations with data */
result = work(me, n, data, tids, nproc);
/* Send result to master */
pvm_initsend(PvmDataDefault);
pvm_pkint(&me, 1, 1);
pvm_pkfloat(&result, 1, 1);
msgtype = 5;
master = pvm_parent();
pvm_send(master, msgtype);
/* Program done. Exit PVM and stop */
pvm_exit();

/* Receive data from master */
rd("init data", ?nproc, ?n, ?data);
/* Do calculations with data */
result = work(id, n, data, tids, nproc);
/* Send result to master */
out("result", id, result);
Global counter in Linda vs. Message Passing

- Example in “Using MPI”, Gropp, et. al. was more than two pages long!

- Several reasons:
  - MPI cannot easily represent data apart from processes
  - Must build a special purpose “counter agent”
  - All data marshalling is done by hand (error prone!)
  - Must worry about “group issues”

- In Linda, counter requires 3 lines of code!
Why Linda’s Tuple Space is important

Parallel program development is much easier than with message passing:

- Dynamic tasking
- Distributed data structures
- Uncoupled programming
- Anonymous communication
- Dynamically varying process pool
Hands on session #4: Monte Carlo PI

- Pi can be calculated by the probability of randomly placed points falling within a circle

- Use master/worker algorithm to parallelize program

\[
\begin{align*}
    a_{\text{square}} &= 4r^2 \\
    a_{\text{circle}} &= \pi r^2 \\
    \pi &= 4 \frac{a_{\text{circle}}}{a_{\text{square}}}
\end{align*}
\]
Poison Pill Termination

- Common Linda idiom in Master/Worker algorithms

- Master creates a special task that causes evaled processes to terminate.

```c
real_main()
{
    for(i=0; i<NWORKERS; i++)
        eval("worker", worker());
    ...
    /* got all results */
    out("task", POISON, t);
    ...
}

worker()
{
    ...
    in("task", ?tid, ?t);
    if (tid==POISON) {
        in("task", ?tid, ?t);
        return();
    }
    ...
}
Hands on session #5: Matrix Multiplication

- This exercise develops a Linda application with non-trivial communication costs
- Write a program that computes $C = A \times B$ where $A$, $B$, $C$ are square matrices
- Parallelism can be defined at any of the following levels:
  - single elements of result matrix $C$
  - single rows (or columns) of $C$
  - groups of rows (or columns) of $C$
  - groups of rows and columns (blocks) of $C$
Hands on session #5: Matrix Multiplication

- For your algorithm, estimate the ratio of communication to computation, assuming that:
  - Computational speed is 100 Mflops
  - Communication speed is 1 Mbytes/sec with 1 msec latency (ethernet)

- How much faster must the network be?