Linda
Parallel programming
Paradigm
Approaches to Parallel Programming

There are two main challenges facing any parallel programmer:
1) How to divide the work among the available processors.
2) Where to store the data and how to get it to processors that need it.

Two radically different approaches to these problems have emerged as the dominant parallel processing paradigms: message passing and distributed data structures.
In message passing parallel programs, there is no Global data.

All data is always held by some process and must be explicitly sent to other processes that need it.

Here each process holds one row of A and computes one element of C while it has the corresponding column of B. The columns of B are passed from process to process to complete.
Distributed Data Structures Matrix Multiplication

workers retrieve tasks from the shared data space, complete them, and then repeat the process until all tasks are done. Here each task is to compute a portion of the result matrix C. Each worker reads the data it needs for its current task—here the relevant portions of A and B—from the global data space, and places its results there when finished as well. The master process, which generated the tasks and placed them into the shared data space, also eventually gathers the results placed there by the worker processes.
Linda—or more precisely, the Linda model—is a general model of parallel computing based on distributed data structures (although, it may be used to implement message passing as well).

Linda calls the shared data space *tuple space*.

Linda programmers don’t need to worry about how tuple space is set up, where it is physically located, or how data moves between it and running processes; all of this is managed by the Linda software system.

Because of this, Linda is logically independent of system architecture, and Linda programs are portable across different architectures, be they shared memory computers, distributed memory computers, or networks of workstations.
Introduction

Why Linda?

Overview

The Linda Model

Master/Worker Model using Virtual Shared Memory

Linda Basics
Introduction

Linda is a proven industry standard for parallel programming and well known for reliability and efficiency. Linda, introduced in the 1980s, is a parallel processing coordination language developed largely by the Linda Group at Yale University, which is a popular system for parallel and distributed computing.

However, traditional Linda implementations have been embedded in host languages via dedicated compilers and preprocessors. Optimization has been, by and large, via static code analysis. This use of pre-compilation has been viewed by Carriero and Gelernter as a distinguishing feature of Linda.
Linda is a reduced set of simple operations intended for distributed and parallel programming.

Linda is based on a **tuple space** memory which supports generative communication.

[http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html](http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html)
Why Linda?

**Portability**
Linda is available on a large number of parallel computer systems, including shared-memory computers, distributed-memory computers, and networks.

Most Linda programs written for one machine run without change on others, facilitating single-source portable parallel codes for all architectures.

**Ease of Use**
Linda implements parallelism via a global, logically-shared memory, using a small number of simple but powerful operations.

The entire approach is easy to understand and quickly mastered.
Investment Preservation

Linda enables you to transform existing sequential C, C++, or Fortran programs into parallel codes quickly and easily.

While costly and specialized parallel computers once were required,

Today you can start out with an existing departmental workstation network, and combine cycles together to achieve parallel performance gains, increased job throughput, and improved resource utilization levels.
Why Linda?(cont.)

**Scalability**
Linda's exceptionally low overhead results in high performance low-level speed from a high-level tool.

**Effective Code Development**
It is possible to optimize code automatically and provide extensive error reporting during the compile and link phases. Debugging is simple and effective.

Linda systems include **Tuplescope**, a graphical debugging tool that makes it easy to debug process management and interprocess communication.

Moreover, both Linda and Tuplescope are fully compatible with standard **UNIX debuggers such as DBX and GDB**.

**Dynamic Load Balancing**
The virtual shared memory data model makes it easy to build software using techniques that lead to automatic load balancing - even among heterogeneous processors.
Overview

• Parallel programming language based on C (C-Linda) and Fortran (Fortran-Linda)

• Combines coordination language of Linda with programming languages of C and Fortran

• Enables users to create parallel programs that perform on wide range of computing platforms

• Based on logically global, associative object memory called tuple space
Overview

• Tuple space provides inter process communication and synchronization logically independent of the underlying computer or network.

• Implements parallelism with a small number of simple operations on tuple space to create and coordinate parallel processes.

• Commercially available from Scientific Computing Associates Inc.
Virtual Shared Memory (VSM)

- Different parts of the data can reside on different processors.
- Looks like one single global memory space to component processes.

Linda's Virtual Shared Memory is known as tuple space

Can be used to implement many different types of algorithms

Lends itself well to master/worker distributed data structure algorithms
Master/Worker Model using Virtual Shared Memory

Master divides work into discrete tasks and puts into global space.

Task and workers are independent of each other.

Workers repeatedly retrieve tasks and put results back into global space.

Workers notified of work completion by having met some condition, receiving a "poison pill" or terminated by some other means.

Master gathers results from global space.
Linda Basics: Definitions

• Tuple Space
  • Linda's name for its shared data space. Tuple space contains tuples.

• Tuples
  • The fundamental data structure of tuple space.
  • Tuples are represented by a list of up to 16 fields, separated by commas and enclosed in parentheses.

  • Examples:
    ('arraydata', dim1, 13, 2)
    (var1, var2)
    ('common block', /datacom/)
    ('array sections', a_array(2:10, 4:8))

• Associative Memory Model
  • A tuple is accessed by specifying its contents.
  • From the programmer's point of view, there is no address associated with a tuple.
Linda Basics: Templates

• Specifies tuple to retrieve
• Consists of sequence of typed fields
• Two kinds of fields
  • Actuals
    • Variables, constants or expression that resolve to constant
  • Formals
    • Holders for data to retrieve.
    • Preceded by a question mark.
    • Assigned values of corresponding fields in matched tuple.
• Example:
  in("arraydata", ?dim1, ?dim2, ?dim3);

  in ("arraydata", 4, ?dim2, ?dim3);

• Both examples will match the tuple put into tuple space with the following out operation:
  out("arraydata", 4, 6, 8);
Linda Basics: basic Operations

Tuple Generation

- **out**
  - Generates a data (passive) tuple
  - Each field is evaluated and put into tuple space
  - Control is then returned to the invoking program
  - Example:
    
    ```
    out ('array data', dim1, dim2)
    ```

- **eval**
  - Generates process (active) tuple
  - Control is immediately returned to invoking program
  - Logically, each field is evaluated concurrently, by a separate process and then placed into tuple space
  - In current implementation, only fields containing function (or subroutine) references result in new processes being created
  - Example:
    
    ```
    eval("test", i, f(i));
    ```
Linda Basics: basic Operations (Cont.)

Tuple Extraction

• **in**
  - Uses a template to retrieve tuple from tuple space.
  - Once retrieved, it is taken out of tuple space and no longer available for other retrievals.
  - If no matching tuple is found process will block. Provides for synchronization between processes.
  - Example:
    
    ```
    in ("arraydata", ?dim1, ?dim2);
    ```

• **rd**
  - Uses a template to copy data without removing it from tuple space.
  - Once read it is still available for others.
  - If no matching tuple is found process will block.
  - Example:
    
    ```
    rd("arraydata", ?dim1, ?dim2);
    ```
Linda Basics: Template Matching Rules

- In order for a template to match a tuple:
  - Should have the same number of fields.
  - Actuals must have same type, length and values as those in corresponding tuple fields.
  - Formals in template must match type and length of corresponding fields in tuple.
- If several tuples match the template, impossible to predict which will be selected.
- The order of evaluation of fields within a tuple or template is undefined. Therefore, the following constructs should be avoided:
  - `out ("string", i++, i);`
    Can't predict whether i will be incremented before or after it is evaluated for the third field.
  - `in("string2", ?j, ?a[j]);`
    Can't predict whether j (in third field) will have value set by ?j (second field) or value before statement was executed.
  - `out('string', x, f(x))`
    In this Fortran example, if the function f() modifies the value of x, we can't predict whether the second field will have the original or the modified value of x.
Examples:
out ('testdata', i, 3, 4+6) will be matched by:

integer cnt, var, sum
character*8 string
.
.
in ('testdata', ?cnt, ?var, ?sum)
or
in ('testdata', ?cnt, ?var, 10)
or
in (?string, ?cnt, ?var, ?sum)
Example Code:

C-Linda - Hello World

File: hello_world.cl

* Simple C Linda example

real_main(int argc, char *argv[])
{
    int nworker, j, hello();
    nworker = atoi(argv[1]);
    for (j = 0; j < nworker; j++)
        eval("worker", hello(j));
    for (j = 0; j < nworker; j++)
        in("done");
    printf("Hello_world is finished.\n");
}

/** subroutine hello **/
int hello (int i)
{
    printf("Hello world from number %d\n", i);
    out("done");
    return(0);
}

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

- Linda operations eval, in, out
- Program is asynchronous
- Master knows it is done when it has retrieved $n$ done tuples from tuple space where $n = \text{number of workers}$
The Linda Model

C-Linda is an implementation of the Linda model using the C programming language, and Fortran-Linda is an implementation of the Linda model using the Fortran programming language.

Since C-Linda is implemented as a precompiler, C-Linda programs are essentially independent of the particular (native) C compiler used for final compilation and linking.
Case Studies

Ray Tracing
Image rendering is a very computationally intensive application. The Rayshade program, written in C, renders color images using ray tracing. It is capable of including texturing effects to simulate different materials in the image, multiple light sources and types (point, spotlight, diffuse), and atmospheric effects like fog or mist.

Matrix Multiplication

Database Searching
This case study looks at a generalized database search program. What the particular data records are is less important than the general issues all such searches raise, many of which are applicable to other types of problems as well.

Molecular Dynamics
Molecular dynamics calculations are performed to simulate the motion within molecules over time. This method is used to study the very large molecules of commercial and biological interest, typically containing thousands of atoms. The atoms in these molecules are constantly in motion; this movement results in changes in the overall molecular structure, which can in turn affect the molecule’s properties. A molecular dynamics simulation calculates the changing structure of a molecule over time in an effort to understand and predict its properties.
Using Linda Code Development System (CDS)

```c
while (1){
    /* Index into my portion of the array that was passed in. Get the chunksize. */
    in ("my_section", ?index, ?chunksize);
}
```
Arrays in C-Linda: Array Specification

In C-Linda operations arrays have the attribute of having a fixed or varying length.

• Fixed Array
  • Has predetermined and unchanging length known at compile time
  • Fixed arrays are specified by using the array name.
  • The following examples puts the entire array into and out of tuple space.
    ```
    int a[20], b[20];
    ...
    out ("fixed array", a);
    in ("fixed array", ?b);
    ```

• Varying Array
  • Uses additional information provided at runtime to specify its current length
    (portion of declared array actually being used).
  • Varying attribute is applied to an array by placing a colon after its name.
  • Information about its current length follows the colon.
**Arrays in C-Linda: Array Specification (Cont.)**

- **Varying Array (cont.)**
  - To specify the **first 10 elements** of \( a \):
    
    \[
    a:10
    \]
  - To specify the **entire array** as a varying array:
    
    \[
    a:
    \]
  - When used as a formal, an integer variable used for the length will be set to the number of elements retrieved.
  - **Example:**
    
    \[
    n ("array", \ ?a:length);
    \]
    
    will have the number of elements retrieved put into the variable \textit{length}.

- **NOTE**: Fixed arrays never match varying arrays.
Some other examples:

- These variables will be used in the examples below:
  ```c
  int *p, a[20], b[20], c[10], len;
  p = a;
  ```

- These *out* operations yield identical tuples:
  ```c
  out("varying array", a:);
  out("varying array", a:20);
  out("varying array", p:20);
  ```
  The length is required in the third case because pointers must always specify an explicit `len`.

- To create a tuple containing the first 10 elements:
  ```c
  out("ten elements", a:10);
  out("ten elements", p:10);
  ```

- The "ten elements" tuple may be retrieved by any one of the following *in* operations:
  ```c
  in("ten elements", ?a:len);
  in("ten elements", ?p:len);
  in("ten elements", ?b:);
  in("ten elements", ?c:);
  ```
  However, `in ("ten elements", ?c)`
  
  would not work, because it is a fixed type and fixed arrays never match variable arrays.
Arrays in C-Linda: Multidimensional Arrays

- Multidimensional arrays match if the portions referred to have the same shape.
- Given the arrays:
  
  ```
  int a[3][5][2], b[4][6][2], c[7][2][5];
  ```
  
- The following would match: (They both point to a fixed array with a length of 2)
  
  ```
  out("multi", a[0][0]);
  in ("multi", ?b[0][0]);
  ```
  
- The following would not match: (The array generated by the out operation is a 5x2, and the in template is a 6x2.)
  
  ```
  out("multi", a[0]);
  in ("multi", ?b[0]);
  ```
  
- Arrays do not have to have the same number of dimensions to match.
  
  - Each set of `in` and `out` operations below match.
    
    ```
    int a[3][5][2], b[5][2], c[2], i;
    out("fixed", a[0]) match with in ("fixed", ?b);
    out("varying", a[0][0]:) match with in ("varying", ?c:);
    out("not an array", a[0][0][0]) match with in ("not an array", ?i);
    ```
Arrays in C-Linda: Template Matching and Arrays

- Arrays match other arrays whose elements are of the same type.
- Fixed arrays match only other fixed arrays of the same length.
- Fixed arrays never match varying arrays even if they are the same size.
- Multidimensional arrays match if the portions referred to have the same shape.
- Actual arrays must agree in both the number of elements and the values of all corresponding elements.
Linda
User’s Guide
&
Reference Manual

Manual version: 6.2
Corresponds to Original and Network Linda version 6.2
May, 2000
The contents of the manual are described briefly below:

Chapter 1: *A Brief Overview of Parallel Programming*
Chapter 2: *Using the Linda Operations*
Chapter 3: *Case Studies*
Chapter 4: *Using Linda on a Network*
Chapter 5: *Debugging Linda Programs*
Chapter 6: *Creating Piranha Programs*
Chapter 7: *Linda Usage and Syntax Summary*
Appendix: *How and Where to Parallelize*
*Bibliography*
Maximum Performance Through Parallel Execution

By overcoming the traditional obstacles to parallel computing - costly hardware, programming difficulty, and lack of code portability - SCIENTIFIC's Linda has become a powerful fulcrum for competitive advantage in many industries, from aerospace and automotive to petroleum, semiconductors, pharmaceuticals, and finance.

Linda provides a simple, yet complete command set which enables process creation, synchronization and communication. Every Linda software system employs powerful application optimization techniques and carefully tuned, architecture-specific run-time systems. Users benefit from, but are not concerned with, low-level details of different machine architectures or communication topologies.

The Linda Model

Linda is a coordination language which supplies the "glue" needed to cement many independent processes together into a single parallel program. Linda provides a virtual shared memory (VSM) that is logically
/* primes.c -- finding prime number with using linda Copyright (C) 1997, 1998, 1999, 2000, 2001, 2002 Wong Weng Fai <wongwf@comp.nus.edu.sg> */

#include "linda.h"
#include <stdio.h>
#include <math.h>
#define MAXVAL 100
#define ANSWER 97

void primes(void)
{
    int me, i, limit, ok;
    linda_in("%s%d", "primeargs", &me);
    limit = sqrt((double) me) + 1;
    for (i = 2; i < limit; i++)
    {
        linda_rd("%s%d%d", "primes", i, &ok);
        if (ok && ((me % i) == 0))
        {
            linda_out("%s%d%d", "primes", me, 0);
            return;
        }
    }
    linda_out("%s%d%d", "primes", me, 1);
}
int main(int argc, char **argv)
{
    int last, i, ok;
    linda_init(NULL);
    for (i = 2; i < MAXVAL; i++)
    {
        linda_out("%s%d", "primeargs", i);
        linda_spawn(primes);
    }

    for (i=2; i < MAXVAL; i++)
    {
        linda_rd("%s%d?d", "primes", i, &ok);
        if (ok == 1)
            last = i;
    }
    linda_end();
    printf("greatest prime is %d\n", last);

    if (ANSWER == last)    return 0;
    else return 1;
}
Functions

G_LOCK_DEFINE_STATIC (linda_threads_slist)
gint linda_spawn_full (GVoidFunc f, gboolean joinable, GError **error)
void linda_thread_join (gpointer data, gpointer user_data)
gpointer linda_void_func_wrapper (gpointer f)
gint linda_do_any (TupleType type, GTimeVal *end_time, const gchar *mask, va_list list)
void linda_init (GThreadFunctions *vtable)
void linda_end (void)
gint linda_spawn (GVoidFunc f)
void linda_out (const gchar *tuple_mask,...)
void linda_rd (const gchar *tuple_mask,...)
gint linda_rdp (const gchar *tuple_mask,...)
void linda_in (const gchar *tuple_mask,...)
gint linda_inp (const gchar *tuple_mask,...)
gint linda_inp_timed (GTimeVal *end_time, const gchar *tuple_mask,...)
gint linda_rdp_timed (GTimeVal *end_time, const gchar *tuple_mask,...)
const gchar *linda_version (void)

Variables

GList *linda_threads_slist = NULL