# pRange Summary

- Binds the work of an algorithm to the data
- **Simplifies** programming task graphs
  - Methods to create tasks
  - Common dependence pattern specifications
  - Compact specification of task dependencies
  - Manages task refinement
  - Simple specification of task graph composition
- Supports multiple programming models
  - Data-parallelism
  - Task-parallelism




# STAPL Example - p_count

**Implementation**

```
template<typename View, typename Predicate>
class p_count_wf {
  //constructor - init member m_pred

  plus<result_type> combine_function(void)
  { return plus<result_type>(); }

  template<typename ViewSet>
  size_t operator()(ViewSet& vs)
  {
    return count_if(vs.sv0().begin(),
           vs.sv0().end(), m_pred);
  }
};

template<typename View, typename Predicate>
p_count_if(View& view, Predicate pred) {
  typedef p_count_wf<View,Predicate>   wf_t;
  wf_t wf(pred);
  return pRange<View, wf_t>(view, wf).execute();
}
```

**Example Usage**

```
stapl_main() {
  p_vector<int>             vals;
  p_vector<int>::view_type  view
    = vals.create_view();

  ... //initialize

  int ret = p_count(view, less_than(5));
}
```

# STAPL Example - p_dot_product

### Implementation

```
template<typename View>
class p_dot_product_wf {
  plus<result_type> get_combine_function(void)
  { return plus<result_type>(); }

  template<typename ViewSet>
  result_type operator()(ViewSet& vs)
  {
    result_type result = 0;
    ViewSet::view0::iterator i = vs.sv0().begin();
    ViewSet::view1::iterator j = vs.sv1().begin();
    for(; i!=vs.sv0.end(); ++i, ++j) {
     result += *i * *j;
  }
};

template<typename View1, typename View2>
p_dot_product(View1& vw1, View2& vw2) {
  typedef p_dot_product_wf<View1, View2>   wf_t;
  wf_t wf;
  return pRange<View1, View2, wf_t>(vw1, vw2, wf).execute();
}
```

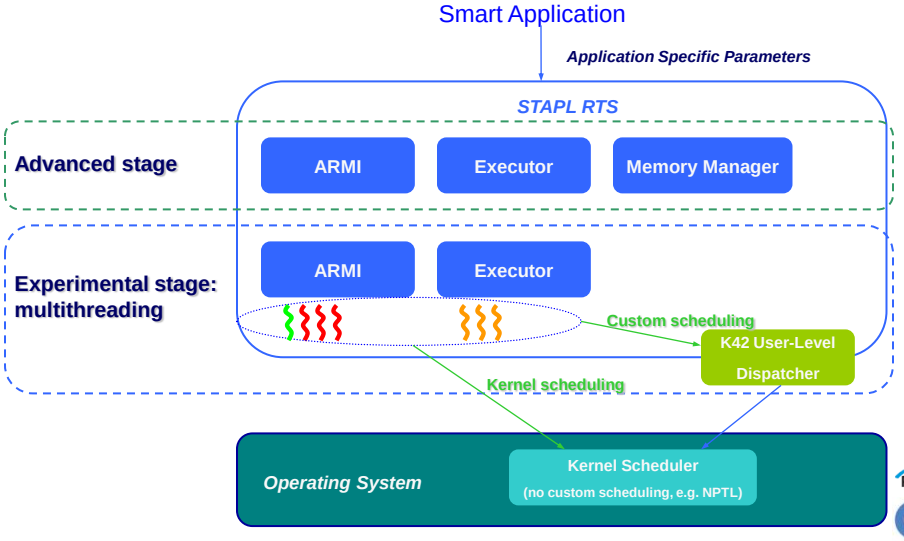
### Example Usage

```
stapl_main() {
  p_vector<int>             vals;
  p_vector<int>::view_type  view1
    = vals.create_view();

  p_vector<int>             more_vals;
  p_vector<int>::view_type  view2
    = more_vals.create_view();

  ... //initialize

  int ret = p_dot_product(view1, view_2);
}
```

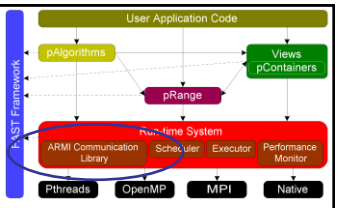



# RTS – Current state

Comm. Thread
RMI Thread
Task Thread




# ARMI – Current State




ARMI: Adaptive Remote Method Invocation

- Abstraction of shared-memory and message passing communication layer (MPI, pThreads, OpenMP, mixed, Converse).
- Programmer expresses fine-grain parallelism that ARMI adaptively coarsens to balance latency versus overhead.
- Support for sync, async, point-to-point and group communication.
- Automated (de)serialization of C++ classes.

ARMI can be as easy/natural as shared memory and as efficient as message passing.




# ARMI Communication Primitives

**Point to Point Communication**

armi_async - non-blocking: doesn't wait for request arrival or completion.

armi_sync - blocking and non-blocking versions.

**Collective Operations**

armi_broadcast, armi_reduce, etc.
can adaptively set groups for communication.

**Synchronization**

armi_fence, armi_barrier - fence implements distributed termination algorithm to ensure that all requests sent, received, and serviced.

armi_wait - blocks until at least at least one request is received and serviced.

armi_flush - empties local send buffer, pushing outstanding to remote destinations.

# RTS – Multithreading (ongoing work)

In ARMI

- Specialized communication thread dedicated the emission and reception of messages
  - Reduces latency, in particular on SYNC requests
- Specialized threads for the processing of RMIs
  - Uncovers additional parallelism (RMIs from different sources can be executed concurrently)
  - Provides a suitable framework for future work on relaxing the consistency model and on the speculative execution of RMIs

In the Executor

- Specialized threads for the execution of tasks
  - Concurrently execute ready tasks from the DDG (when all dependencies are satisfied)




# RTS Consistency Models

Processor Consistency (default)

- Accesses from a processor on another's memory are sequential
- Requires in-order processing of RMIs
  - Limited parallelism

Object Consistency

- Accesses to different objects can happen out of order
- Uncovers fine-grained parallelism
  - Accesses to different objects are concurrent
  - Potential gain in scalability
- Can be made default for specific computational phases

Mixed Consistency

- Use Object Consistency on select objects
  - Selection of objects fit for this model can be:
    - Elective – the application can specify that an object's state does not depend on others' states.
    - Detected – if it is possible to assert the absence of such dependencies
- Use Processor Consistency on the rest

# RTS Executor

Customized task scheduling

- Executor maintains a ready queue (all tasks for which dependencies are satisfied in the DDG)
- Order tasks from the ready queue based on a scheduling policy (e.g. round robin, static block or interleaved block scheduling, dynamic scheduling …)
- The RTS decides the policy, but the user can also specify it himself
- Policies can differ for every pRange

Customized load balancing

- Implement load balancing strategies (e.g. work stealing)
- Allow the user to choose the strategy
- K42 : generate a customized work migration manager




# RTS Synchronization

- Efficient implementation of synchronization primitives is crucial
  - One of the main performance bottlenecks in parallel computing
  - Common scalability limitation

Fence

- Efficient implementation using a novel Distributed Termination Detection algorithm

Global Distributed Locks

- Symmetrical implementation to avoid contention
- Support for logically recursive locks (required by the compositional SmartApps framework)

Group-based synchronization

- Allows efficient usage of ad-hoc computation groups
- Semantic equivalent of the global primitives
- Scalability requirement for large-scale systems

# Productivity

- Implicit parallelism
- Implicit synchronizations/communications
- Composable (closed under composition)
- Reusable (library)
- Tunable by experts (library not language)
- Compiles with any C++ compiler (GCC)
- Optionally exposes machine info
- Shared Memory view for user
- High level of abstraction – Generic Programming




# Performance

- Latency reduction: Locales , data distribution
- Latency Hiding: RMI, multithreading, Asynch Communications
- Optionally exposes machine info
- Manually tunable for experts
- Adaptivity to input and machine (machine learning)

# Portability

- Library – no need for special compiler
- RTS needs to be ported – not much else
- High level of abstraction




# References

Cilk

http://supertech.csail.mit.edu/cilk/

http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

Dag-Consistent Distributed Shared Memory, Blumofe, Frigo, Joerg, Leiserson, and Randall, In 10th International Parallel Processing Symposium (IPPS '96), April 15-19, 1996, Honolulu, Hawaii, pp. 132-141.

TBB

http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm

TBB Reference Manual – provided with package

# References

- HPF
  - HPFF Homepage - http://hpff.rice.edu/
  - High Performance Fortran: history, overview and current developments. H Richardson, Tech. Rep. TMC-261, Thinking Machines Corporation, April 1996.
- Chapel
  - http://chapel.cs.washington.edu/
  - Chapel Draft Language Specification. http://chapel.cs.washington.edu/spec-0.702.pdf
  - An Introduction to Chapel: Cray's High-Productivity Language. http://chapel.cs.washington.edu/ChapelForAHPCRC.pdf




# References

- Fortress
  - http://research.sun.com/projects/plrg
  - Fortress Language Specification. http://research.sun.com/projects/plrg/fortress.pdf
  - Parallel Programming and Parallel Abstractions in Fortress. Guy Steele. http://irbseminars.intel-research.net/GuySteele.pdf
- Stapl
  - http://parasol.tamu.edu/groups/rwergergroup/research/stapl
  - A Framework for Adaptive Algorithm Selection in STAPL, Thomas, Tanase, Tkachyshyn, Perdue, Amato, Rauchwerger, In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, pp. 277-288, Chicago, Illinois, Jun 2005.

# Table of Contents






# UPC

- Unified Parallel C
- An explicit parallel extension of ISO C
- A partitioned shared memory parallel programming language
- Similar to the C language philosophy
  - Programmers are clever

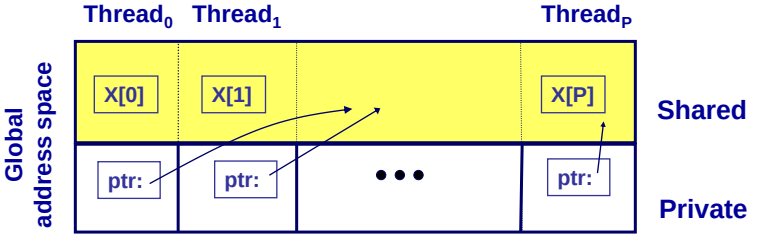
Adapted from http://www.upc.mtu.edu/SC05-tutorial

# Execution Model

- UPC is SPMD
  - Number of threads specified at compile-time or run-time;
  - Available as program variable **`THREADS`**
  - **`MYTHREAD`** specifies thread index (**`0..THREADS-1`**)
- There are two compilation modes
  - Static Threads mode:
    - THREADS is specified at compile time by the user
    - THREADS as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads




# UPC is PGAS




- The languages share the global address space abstraction
  - Programmer sees a single address space
  - Memory is logically partitioned by processors
  - There are only two types of references: local and remote
  - One-sided communication

# Hello World

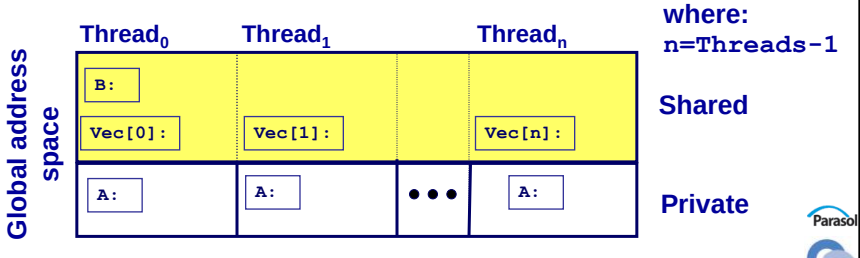
- Any legal C program is also a legal UPC program
- UPC with P threads will run P copies of the program.
- Multiple threads view

```
#include <upc.h>  /* UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC
  world\n", \
         MYTHREAD, THREADS);
}
```




# Private vs. Shared Variables

- Private scalars (`int A`)
- Shared scalars (`shared int B`)
- Shared arrays (`shared int Vec[TREADS]`)
- Shared Scalars are always in threads 0 space
- A variable local to a thread is said to be **affine** to that thread




# Data Distribution in UPC

- Default is cyclic distribution
  - `shared int V1[N]`
  - Element `i` affine to thread `i%THREADS`
- Blocked distribution can be specified
  - `shared [K] int V2[N]`
  - Element `i` affine to thread `(N/K)%THREADS`
- Indefinite ()
  - `shared [0] int V4[4]`
  - all elements in one thread
- Multi dimensional are linearized according to C layout and then previous rules applied




# Work Distribution in UPC

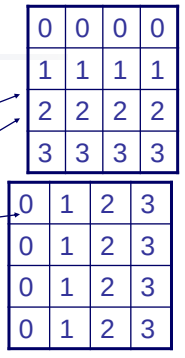
- UPC adds a special type of loop

```
upc_forall(init; test; loop; affinity)
   statement;
```

- Affinity does not impact correctness but only performance
- Affinity decides which iterations to run on each thread. It may have one of two types:
  - Integer: `affinity%THREADS` is `MYTHREAD`
  - E.g., `upc_forall(i=0; i<N; i++; i)`
  - Pointer: `upc_threadof(affinity)` is `MYTHREAD`
  - E.g., `upc_forall(i=0; i<N; i++; &vec[i])`

# UPC Matrix Multiply

```
#define N 4
#define P 4
#define M 4
// Row-wise blocking:
shared [N*P/THREADS] int a[N][P], c[N][M];
// Column-wise blocking:
shared[M/THREADS] int b[P][M];

void main (void) {
  int i, j , l; // private variables

  upc_forall(i = 0 ; i<N ; i++; &c[i][0])
    for (j=0 ; j<M ;j++) {
      c[i][j] = 0;
      for (l= 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b[l][j];
    }
}
```




Replicating `b` among processors would improve performance




# Synchronization and Locking

- Synchronization
  - Barrier: block until all other threads arrive
    - `upc_barrier`
  - Split-phase barriers
    - `upc_notify` this thread is ready for barrier
    - `upc_wait` wait for others to be ready
- Locks: `upc_lock_t`
  - Use to enclose critical regions
    - `void upc_lock(upc_lock_t *l)`
    - `void upc_unlock(upc_lock_t *l)`
  - Lock must be allocated before use

# Collectives

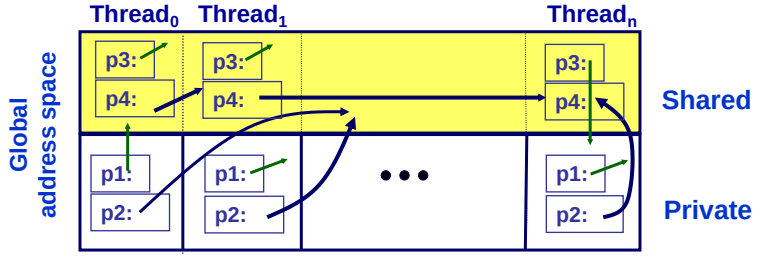
- Must be called by all the threads with same parameters
- Two types of collectives
  - Data movement: scatter, gather, broadcast,…
  - Computation: reduce, prefix, …
- When completed the threads are synchronized
- E.g.,

```
res=bupc_allv_reduce(int, in, 0, UPC_ADD);
```

Type Input Thread Operation



# UPC Pointers




```
int *p1;         /* private pointer to local memory */
shared int *p2; /* private pointer to shared space */
int *shared p3; /* shared pointer to local memory */
shared int  *shared p4; /* shared pointer to
                           shared space */
```

- Pointers-to-shared are more costly to dereference
- The use of shared pointers to local memory are discouraged

# Memory Consistency

- UPC has two types of accesses:
  - Strict: Will always appear in order
  - Relaxed: May appear out of order to other threads
- There are several ways of designating the type, commonly:
  - Use the include file:

```
#include <upc_relaxed.h>
```

  - All accesses in the program unit relaxed by default
  - Use strict on variables that are used as synchronization (`strict shared int flag;`)

```
data = …                while (!flag) { };
flag = 1;               … = data;   // use the data
```




# Additional Features

- Latency management: two levels of proximity exposed to the user
- Portability: UPC compilers are available for many different architectures
- Productivity: UPC is a low-level language, the main objective is performance

# Table of Contents



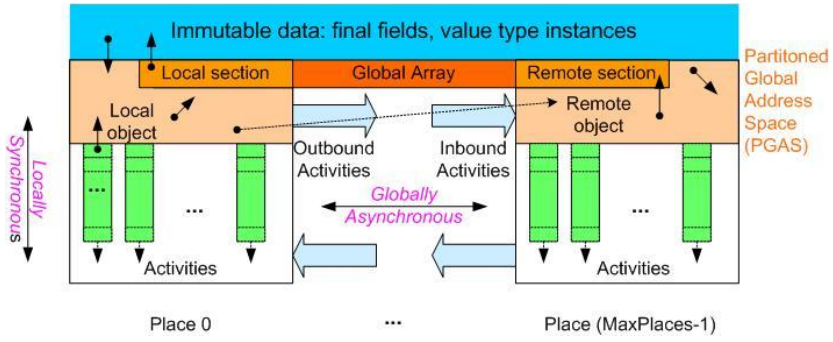



# X10

- Developed by IBM as part of DARPA HPCS
- Draws from Java syntax and arrays in ZPL
- Partitioned Global Address Space (PGAS)
- Clocks - generalized barrier synchronization
- Constructs for explicit data and work placement

**Adapted from presentations at:** http://x10.codehaus.org/Presentations

# The X10 Model




**Place** - collection of resident activities & objects (e.g., SMP node of cluster).

**Activities** - lightweight thread of execution.

**Locality Rule**
Access to data must be performed by a local activity. Remote data accessed by creating remote activities

**Ordering Constraints (Memory Model)**
Locally Synchronous:
Guaranteed coherence for local heap.
Strict, near sequential consistency.

Globally Asynchronous:
No ordering of inter-place activities.
Explicit synchronization for coherence.



# The X10 Model

## Execution Model

- Explicit data parallelism, **`foreach`**
- Explicit task parallelism **`future, async`**
- Explicit, asynchronous, one-sided communication with **`future`**
- Explicit synchronization
  - **`clock`**, **`finish`**, **`future`**, **`atomic`** section (within a place)
- Multi-level memory model under development
  - Within a place - more strict, not quite sequential consistency
  - Across places - relaxed, explicit synchronization required

# X10 - Regions

- Defines a set of *points* (indices)
  - Analogous to Chapel domains
  - User defined regions in development

```
region Null = [];  // Empty 0-dimensional region
region R1 = 1:100; // 1-dim region with extent 1..100.
region R1 = [1:100]; // Same as above.
region R2 = [0:99, -1:MAX_HEIGHT];
region R3 = region.factory.upperTriangular(N);
region R4 = region.factory.banded(N, K);
   // A square region.
region R5 = [E, E];
   // Same region as above.
region R6 = [100, 100];
   // Represents the intersection of two regions
```




# X10 - Distributions

- Maps every point in a region to a place
  - Analogous to Chapel distributed domains
  - User distributions regions in development

```
dist D1 = dist.factory.constant(R, here);//maps region R to local place
dist D2 = dist.factory.block(R);       //blocked distribution
dist D3 = dist.factory.cyclic(R);      //cyclic distribution
dist D4 = dist.factory.unique();       //identity map on [0:MAX_PLACES-1]

double[D] vals;
vals.distribution[i] //returns place where ith element is located.
```

# X10 - Data Parallelism

**`[finish] foreach(i : Region) S`**

*Create a new activity at place P for each point in Region and execute statement S. Finish forces termination synchronization.*

```
public class HelloWorld2 {
  public static void main(String[] args) {
    foreach (point [p] : [1:2])
      System.out.println("Hello from activity " + p + "!");
  }
}
```




# X10 - Data Parallelism

**`[finish] ateach(i : Distribution) S`**

*Create a new activity at each point in Region at the place where it is mapped in the Distribution. Finish forces termination synchronization.*

```
public class HelloWorld2 {
  public static void main(String[] args) {
      ateach (place p: dist.factory.unique(place.MAX_PLACES))
      System.out.println("Hello from place " + p + "!");
  }
}
```

# X10 - Task Parallelism

**`[finish] async(P) S`**

*Create a new activity at place P, that executes statement S.*

```
//global array
double a[100] =  …;
int k = …;

async (3) {
    // executed place 3
    a[99] = k;
}

//continue without waiting
```

```
//global array
double a[100] =  …;
int k = …;

finish async (3) {
    // executed place 3
    a[99] = k;
}

//wait for remote completion
```




# X10 - Task Parallelism

**`future(P) S`**

*Similar to* **`async`***, returns result from remote computation.*

```
// global array
final double a[100] =  …;
final int idx = …;

future<double> fd =
  future (3)
  {
    // executed at place 3
      a[idx];
  };

int val = fd.force(); //wait for fd
completion
```

# X10 - Synchronization

- Atomic block
  - conceptually executed in a single step while other activities are suspended
  - must be nonblocking, no task spawning (e.g., no communication with another place)

```
// push data onto concurrent
// list-stack
Node node = new Node(data);
atomic {
      node.next = head;
      head = node;
}
```




# X10 - Synchronization

- Clocks
  - Generalization of barrier
    - Defines program phases for a group of activities
    - Activities cannot move to next phase until all have acquiesced with a call to **`next`**
  - Activities can register with multiple clocks
  - Guaranteed to be deadlock free
  - **`next, suspend, resume, drop`**

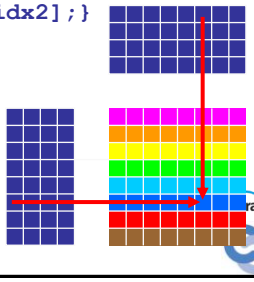
# X10 - Synchronization

```
final clock c = clock.factory.clock();
foreach (point[i]: [1:N]) clocked (c) {
    while ( true ) {
      //phase 1
      next;
      //phase 2
      next;
      if ( cond )
         break;
    } // while
 } // foreach
 c.drop();
```




# X10 - Matrix Multiply

```
double[.] A = new double[D1]; //defined on Region R1
double[.] B = new double[D2]; //defined on Region R2
double[.] C = new double[D3]; //defined on Region R3
...
finish ateach(point ij : D3) {
   for(point k : R1[1]) {
     point idx1 = new point(ij[0],k);
     point idx2 = new point(k, ij[1]);
     future<double> a(A[idx1].location) {A[idx1];}
     future<double> b(B[idx2].location) {B[idx2];}
     C[i] += a.force() * b.force();
   }
}
```



# X10 - Productivity

- New programming language based on Java
- Abstraction
  - Relatively low for communication and synchronization
  - Transparency was a design goal
- Component reuse
  - Java style OOP and interfaces
  - Generic types and type inference under development




# X10 - Productivity

- Tunability
  - Implementation refinement via Distributions and work placement
- Defect management
  - Reduction with garbage collection
  - Detection and removal with integration with Eclipse toolkit
- Interoperability
  - C library linkage supported, working on Java

# X10 - Performance

- Latency Management
  - Reducing
    - Data placement - distributions.
    - Work placement - `ateach, future, async`
  - Hiding
    - Asynchronous communication with `future`
    - Processor virtualization with activities
- Load Balancing
  - Runtime can schedule activities within a place




# X10 - Portability

- Language based solution, requires compiler
- Runtime system not discussed. Must handle threading and communication - assumed to be part of model implementation
- **places** machine information available to programmer
- Parallel model not effected by underlying machine
- I/O not addressed in standard yet

# References

- UPC
  - http://upc.gwu.edu/
  - http://www.upc.mtu.edu/SC05-tutorial
- X10
  - http://x10.codehaus.org/Presentations
  - http://x10.codehaus.org/Tutorials




# Table of Contents



# Linda

- History
  - Developed from 1992 by N. Carriero and D. Gelernter
  - A Commercial version is provided by Scientific Computing Associates, Inc.
  - Variations: TSpace (IBM), JavaSpaces (SUN)
- Programming Style
  - Processes creation is implicit
  - Parallel processes operate on objects stored in and retrieved from a shared, virtual, associative memory (**Tuple Space**)
  - Producer-Consumer approach

*Adapted from http://www.lindaspaces.com/teachingmaterial/LindaTutorial_Jan2006.pdf*




# Linda

- Productivity
  - Linda extends traditional languages (C, Java,…)
  - The abstraction provided is intuitive for some class of problems
  - Object stored in the Tuple Space has a global scope: the user have to take care of associates the right keys
- Portability
  - Tuple Space has to be implemented
  - Code analysis is architecture dependent
  - If objects in the shared space contains references to values a shared memory has to be provided

# Linda

- Performance
  - Depends on Tuple Space implementation
    - Architecture is hidden to the user
  - Code analysis can provide optimizations
- Defect analysis
  - Commercial implementation provides debuggers and profilers




# Tuple Space

- A Tuple is a sequence of typed fields:
  - (“Linda”, 2, 32.5, 62)
  - (1,2, “A string”, a:20) // array with size
  - (“Spawn”, i, f(i))
- A Tuple Space is a repository of tuples
- Provide:
  - Process creation
  - Synchronization
  - Data communication
  - Platform independence

# Linda Operations (read)

- Extraction
  - `in("tuple", field1, field2);`
    - Take and remove a tuple from the tuple space
    - Block if the tuple is not found
  - `rd("tuple", field1, field2);`
    - Take a tuple from the space but don't remove it
    - Block if the tuple is not found
  - `inp`, `rdp`: as in and rd but non-blocking




# Linda Operations (write)

- Generation
  - `out("tuple", i, f(i));`
    - Add a tuple to the tuple space
    - Arguments are evaluated before addition
  - `eval("tuple", i, f(i));`
    - A new process compute f(i) and insert the tuple as the function returns
    - Used for process creation

# Tuple matching

- Tuples are retrieved by matching
  - `out("Hello", 100)`
  - `in("Hello", 100) // match the tuple`
  - `in("Hello", ?i) // i=100`
- Tuples matching is non-deterministic
  - `out("Hello", 100)`
  - `out("Hello", 99)`
  - `in("Hello", ?i) // i=99 or i=100`
- Tuple and template must have the same number of fields and the same types




# Atomicity

- The six Linda operations are atomic
  - A simple counter

```
in("counter", ?count);
out("counter", count+1);
```

  - The first operation remove the tuple gaining exclusive access to the counter
  - The second operation release the counter

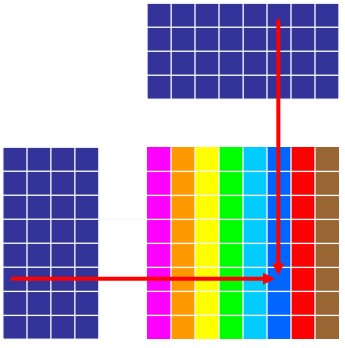
# Hello world

```
linda_main(int i) {
  out("count", 0);
  for(int i=1; i<=NUM_PROCS; i++)
      eval("worker",hello_world(i));
  in("count", NUM_PROCS);
  printf("All processes done.\n");
}

void hello_world (int i) {
  int j;
  in("count", ?j); out("count", j+1);
  printf("Hello world from process %d,", i);
  printf(" count %d\n", j);
}
```






# Matrix Multiply



```
for(int i=0; i<M; ++i) {
  for(int k=0; k<L; ++k) {
    for(int j=0; j<N; ++j) {
      C[i][j] =
          A[i][k]*B[k][j];
    }
  }
}
```

**A parallel specification:**
**$C_{ij}$ is the dot-product of row *i* of *A* and column *j* of *B***

# Matrix Multiply in Linda

```
Void // Compute C=A*transpose(B)
matrix_multiply(double A[m][n],B[l][n],C[m][l]) {
  for (int i=0; i < m; i++) // Spawn internal products
      for (int j=0; i < l; j++) {
            ID = i*n + j;
            eval("dot", ID, \
                 dot_product(&A[i], &B[j], ID));
      }
  for (int i=0; i < n; i++) // Collect results
      for (int j=0; j < n; j++) {
            ID = i*n + j;
            in("dot", ID, ?C[i][j]);
      }
}
```




# Matrix Multiply in Linda (2)

```
double dot_product(double A[n],\
            double B[n], int ID) {
  // ID is not used in the
  // sequential version of dot_product
  double sum=0;
  for (int i=0; i<m; i++)
          sum += A[i]*B[i];
  return sum;
}
```

# Parallel dot-product

```
double dot_product(double *A, double *B, int ID) {
   double p;
   for (int i=0 ; i < m ; i++)
        eval("p-dot", ID, p_prod(A,B,i*(n/m),(n/m)));
   sum = 0;
   for (int i=0 ; i < m ; i++) {
        in("p-dot", ID, ?p);
        sum += p ;
   }
   return sum ;
}
double p_prod(double *A,double *B,int start, int len) {
   double sum = 0;
   for (int i=start; i < len+start; i++)
        sum += A[i]*B[i];
   return sum;
}
```




# Nested Parallelism

- Matrix multiply uses nested parallelism
- Tuples of dot_product have the same types as tuples in matrix_multiply but they have a different string identifier
  - ("dot", int, double*)
  - ("p-dot", int, double*)
- Correctness is guaranteed by ID and commutativity of addition

# Runtime

- Tuple rehashing
  - Runtime observe patterns of usage, remaps tuple to locations
    - Domain decomposition
    - Result tuples
    - Owner compute
- Long fields handling
  - Usually long fields are not used for mathcing
  - Bulk transfer
- Knowing implementation and architecture details and helps in optimizing user code




# Table of Contents



# MapReduce

- Used by Google for production software
- Used on 1000s processors machines
- Automatic parallelization and distribution
- Fault-tolerance
- It is a library built in C++

*Adapted From: http://labs.google.com/papers/mapreduce.html*




# MapReduce Model

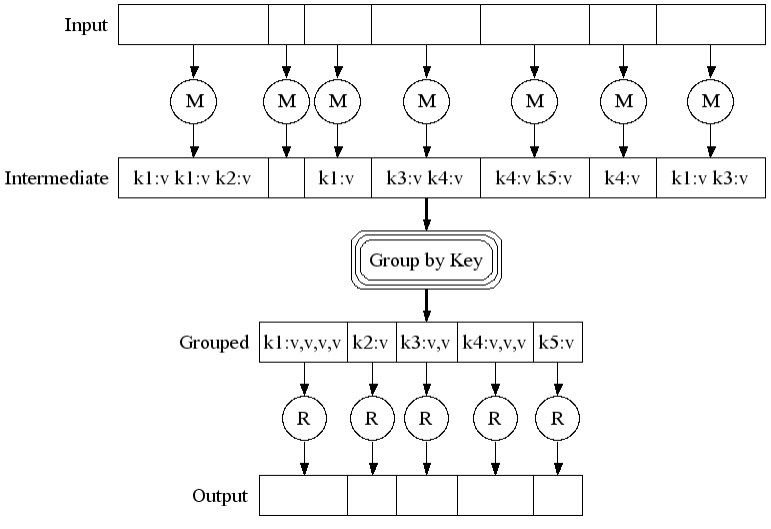
- Input & Output are sets of key/value pairs
- Programmer specifies two functions:
  - `map(in_key, in_value) -> list(out_key, intermediate_value)`
    - Processes input key/value pair
    - Produces set of intermediate pairs
  - `reduce(out_key, list(intermediate_value)) -> list(out_value)`
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values (usually just one)

# Example: Word Count

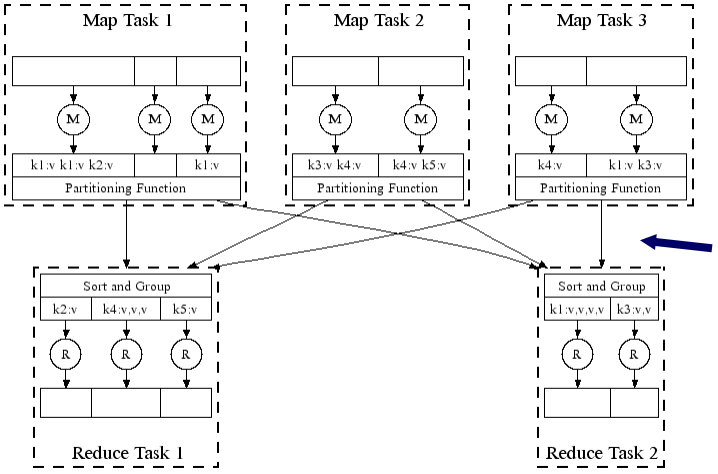
```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
      EmitIntermediate(w, "1");

reduce(String output_key, Iterator
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0; for each v in
  intermediate_values: result += ParseInt(v);
  Emit(AsString(result));
```




# Sequential Execution Model




# Parallel Execution Model






# Parallel Execution Model

- Fine granularity tasks: many more map tasks than machines
- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing
- Often use 200,000 map/5000 reduce tasks w/ 2000 machines

# Performance

- Typical cluster:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - distributed file system manages data (GFS)
  - Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines




# Performance: Locality

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect: Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate

# Performance: Replication

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time




# Performance

- Sorting guarantees within each reduce partition
- Compression of intermediate data
- Combiner: useful for saving network bandwidth

# Fault Tolerance

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master
- Master failure not handled yet
- Robust: lost 1600 of 1800 machines once, but finished fine




# Productivity

- User specifies only two functions
- May be complex to specify a general algorithm
- Highly productive for specific kind of problems

# Table of Contents

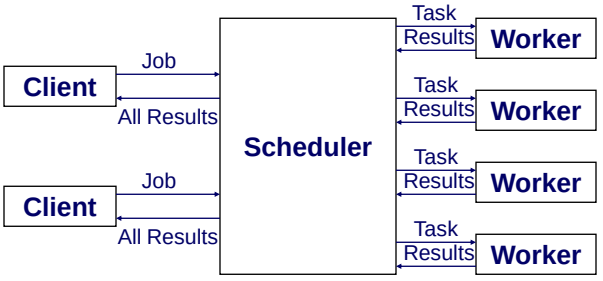





# MATLAB DCE – Distributed Computing Server

- Executing independent jobs in a cluster environment
- A job is a set of tasks
- A task specifies input data and operations to be performed
- A scheduler takes a job and executes its tasks

# Execution Model







# Job Creation and Execution

- Create a Scheduler: `sched = findResource('scheduler', 'type', 'local')`
- Create a Job: `j = createJob(sched);`
- Create Tasks
  - `createTask(j, @sum, 1, {[1 1]});`
  - `createTask(j, @sum, 1, {[2 2]});`
- Submit job: `submit(j);`
- Get results
  - `waitForState(j);`
  - `results = getAllOutputArguments(j)`

```
results =
[2]
[4]
```

- Destroy job: `destroy(j);`

*Number of output arguments*

# Portability

- Different ways to pass data to workers
  - Passing paths for data and functions when using a shared file system
  - Compressing and passing data and functions to workers initializing an environment at worker place
- The first way is less portable even though more efficient




# Productivity

- MATLAB DCE is a queuing system
- Schedule independent jobs
- It may be difficult to code an arbitrary parallel algorithm
- Good for speeding up huge computation with very high level independent tasks

# References

- Linda
  - http://www.lindaspaces.com/about/index.html
  - http://www.almaden.ibm.com/cs/TSpaces/
  - http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/
- MapReduce
  - http://labs.google.com/papers/mapreduce.html
  - http://www.cs.virginia.edu/~pact2006/program/mapreduce-pact06-keynote.pdf
- MATLAB DCE
  - http://www.mathworks.com/products/distriben/
  - http://www.mathworks.com/products/distribtb/




# PP Lecture Conclusions

- High level PPM – high productivity?
- Low level PPM – high performance?
- Safety in higher abstraction
- Needed: Parallel RTS, Debuggers

- Desperately Needed:

  **State of the art Compilers & Tools**