

Parallel extension language Linda

Name

hp-linda : A free High Performance implementation of the parallel extension language Linda for Amoeba.

(Linda ia a registered trademark of Scientific Computing Associates)

Synopsis

```
#include "linda.h"

errstat linda_init (char *pname);
errstat linda_start_svr ( char *tsname, char *host);
errstat linda_spawn ( char *fname, char *host);
int linda_main ( int argc, char **argv);

int Linda_worker_id;
char *Linda_my_name;

LINDA_MASTER (master func);
LINDA_FUNC (worker function);

errstat out ( char *tuple_mask, ... );
errstat in ( char *tuple_mask, ... );
errstat rd ( char *tuple_mask, ... );

errstat linda_init_ts ();
int linda_main_ts ( int argc, char **argv);

errstat out_ts ( char *tsname, char *tuple_mask, ... );
errstat in_ts ( char *tsname, char *tuple_mask, ... );
errstat rd_ts ( char *tsname, char *tuple_mask, ... );
```

Description

Linda is a **parallel coordination language** developed largely by the Linda Group at Yale University, which is a popular system for parallel and distributed computing - mainly in the UNIX domain. Original used in conjunction with a pre-compiler system, however, Linda implementations have been built avoiding the need for pre-compilation in form of generic library calls, for example p4-Linda.

HP-Linda follows this idea, too.

The Linda model is a parallel programming model which uses a so called global **tuple space** to coordinate the passing of messages and therefore data between processes. The tuple space is a shared data space and contains various **tuples**, the fundamental data structure of a tuple space.

Examples for tuples:

```
('myarraydata', intarr, 100, 200)
(intvar1, intvar2)
('mutex key', muval)
```

Tuples are represented by a list of argument fields of various types, for example strings, integers, integer arrays, floats and so on.

Tuples are managed by the **Tuple server**, a separate process.

Tuple operations

They consist of only six functions, those being:

```
out produces a tuple
in consumes a tuple
rd reads a tuple

inp consumes a tuple if available
rdp reads a tuple if available

eval executes a function in a subprocess
      producing a tuple
```

Tuple Generation

OUT operation

- * Generates a data (passive) tuple
- * Each field is evaluated and put into the tuple space
- * Control is then returned to the invoking program

* Example:

```
dim1=100
dim2=200

out('array data', dim1, dim2)
```

Tuple Extraction

IN operation

* Uses a template (or anti tuple) to retrieve a tuple from the tuple space. The template consists of actuals (matching data) and formals (data to be retrieved), indicated with a '?'.
* Once retrieved, it is taken out of the 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);
```

```
in("arraydata",100,?dim2);
```

The (integer) variables contain the values dim1=100 and dim2=200.

RD operation

* Uses a template to copy data without removing it from the tuple space. The template consists of actuals (matching data) and formals (data to be retrieved), indicated with a '?'.
* Once read it is still available for others.

* If no matching tuple is found, process will block.

* Example:

```
rd("arraydata",?dim1,?dim2);
```

```
rd("arraydata",?dim1,200);
```

The (integer) variables contain the values dim1=100 and dim2=200.

Template Matching Rules

In order for a template to match a tuple:

* Have to have the same number of fields

* Actuals must have the same type, length and values as those in corresponding tuple fields

* Formals in the template must match type and length of corresponding fields of tuple

* Example:

```
The tuple
("string",100,int array(100))

matches with the templates (anti tuples):

("string",100,?int array(100))
("string",?int,?int array(100))

but not with

("string",111,?int array(100))
("moreisless",100,?int array(100))
("string",3.1454,?int array(100))
```

If several tuples match the template, the first one found will be taken from the tuple space.

HP-Linda implementation

In *hp-linda*, there are some restrictions to the generic Linda model. First, no *eval* operation exists. This operation was replaced by the *linda_spawn* function.

All linda operations are performed with a single Tuple Space server **linda[U]**. But several different tuple spaces can exist in parallel, each with its own tuple space server. Each tuple space has a unique name from which the server port is derived.

Hp-linda is a pure library implementation. Therefore, an additional **tuple mask string** is needed for each tuple operation to determine the type of a tuple argument. It's always the first argument of a linda operation. More details in the Programming Interface.

Only the blocking operations are implemented. Because the *hp-linda* interface can be used concurrently by many threads (up to `TS_MAX_THREADS`) in one program, there is no need for the non blocking counterparts and polling abuse is avoided.

In an *IN* operation, all aggregates (arrays) are treated always as formals!

HP-Linda supports operations in different tuple spaces within the same program (distributed tuple space).

Programming Interface

There are two possible cases to perform linda's master/worker modell:

1. All sub parts are splitted in different programs and started by hand:
 - One Master program
 - One Linda tuple space server
 - N Worker programs
2. The Master and Worker functions are glued in one program, and only one program must be started. This program does automatically the worker replication and starts the linda server.

1. Simple Linda Master/Worker modell

Before the Linda interface can be used, the user must choose a unique tuple space name, for example "mylinda". It's necessary to start a tuple space server **linda[U]** :

```
linda -n "mylinda"
```

To prepare the client interface, both in the worker and the master program, you must call

```
errstat  
linda_init(pname)  
char *pname
```

where *pname* is the above chosen tuple space name. From this name string, a client/server port will be build for the client-server communication. Now you're able to use the linda tuple operations.

2. All in One Linda Master/Worker modell

Only one program must be compiled and started. The master and worker function(s) are part of this linda program. Additional, all initiation is done by the the function

```
int  
linda_main(argc, argv)  
int argc;  
char **argv;
```

This function needs the *argc* and *argv* arguments from the program *main(argc, argv)* entry routine. *Linda_main* is called both by the master and the workers and will perform the following steps:

1. Scan the command line arguments.
2. Decide if it was called by the master or worker.

Master:

3. Call *linda_scan_pooldir()* to get a list of available hosts used for worker spawning and for the linda tuple space server. If no one is specified, use the user's default run server (*DEFAULT_POOLDIR*).
4. Call *linda_init()* to perform initial client setup.
5. Call *linda_checkpoint_init()* to initialize the process checkpointing.
6. Start linda tuple space server either on user specified host or a host selected from the host pool list (with *linda_find_next_host()*). Except in the case the command line argument "-s -" exists: no server will be started.
7. Call the master function specified with *LINDA_MASTER()*.

Worker:

3. Do initial linda client setup, call *linda_init()*.
4. Call the desired worker function, specified by *LINDA_FUNC()* and the command line arguments

```
-f <function name>  
-w <worker id>
```

The behaviour of *linda_main* is mainly controlled by the arguments given to the program command line. Those are:

-n <tuple space name>	The name of the Tuple space used by the workers and the master.
-N <num workers>	The number of workers which will be started by <code>linda_main()</code>
-r <pool dir>	The run server pool directory
-s <server host>	Use <server host> for the linda tuple space server instead selecting one from the pool dir
-v	Verbose mode: worker, master, linda server
--	After this argument, only user supplied arguments; ignored by <code>linda_main()</code>

The master function to be called must be declared with

```
LINDA_MASTER(<Master function name>);
```

before the `linda_main()` call appears!

Additionally, all worker function needed in the linda program must be declared before `linda_main()`, too:

```
LINDA_FUNC(<Worker function name>);
```

A typical linda program prototype:

```
int
main(int argc, char **argv)
{
    LINDA_FUNC(myworker);
    LINDA_MASTER(mymaster);
    return linda_main(argc, argv);
}

void
mymaster(int argc, char **argv)
```

```
{
    ...
}
void
myworker()
{
    ....
}
```

The master function gets the pointer argument array *argv* of all user arguments, given after the option "--" and *argc* reflects the number of user arguments.

Somewhere in the master control function, the worker programs can be started with

```
errstat
linda_spawn(funcname,host)
char *funcname;
char *host;
```

Linda_spawn starts a new process, and the worker function *funcname* will be called in the new spawned process. The *host* argument is optionally. If *host* is a NULL pointer, the host will be chosen in round-robin fashion from the host pool list previously scanned from the specified (or default) run server. The number of desired workers to be started are determined either statically by the master function, or dynamically by the command line argument "-N ##" and can be retrieved by the global variable

```
int Linda_worker_num;
```

The worker can retrieve his identity number from the global variable

```
int Linda_my_id;
```

The first worker get the id number 0. The master program get the ownership of the spawned processes. Normal and unnormal termination is observed by the master. On interrupt and exit, the master try to terminate all not terminated childs: the linda server and still running workers.

Tuple Out operation:

```
errstat
out(tuple_mask,...)
char *tuple_mask
```

Tuple In and Rd operation:

```
errstat
in(tuple_mask,...)
char *tuple_mask
```

```
errstat
rd(tuple_mask,...)
char *tuple_mask
```

The IN/RD tuple space operations are performed in blocking mode. If no matching tuple is available, the operation blocks until a matching tuple arrives, or an interrupt (User interrupt) occurred.

3. Multiple Tuple Spaces

Additionally to the single tuple space model, it's possible to use several different tuple spaces within the same Linda program with different tuple space servers. With this multi space model, it's possible to simply build up a distributed tuple space.

Tuple Out operation:

```
errstat
out_ts(tsname, tuple_mask,...)
char *tsname;
char *tuple_mask;
```

Tuple In and Rd operation:

```
errstat
in_ts(tsname,tuple_mask,...)
char *tsname;
char *tuple_mask;
```

```
errstat
rd_ts(tsname,tuple_mask,...)
char *tsname;
char *tuple_mask;
```

Tsname is the user chosen name for the specific tuple space.

Warning: Never mix single and multiple space operations in one program. You will fail!

For the multi tuple space there are different initialization functions:

All initiation is done by the the function

```
int  
linda_main_ts(argc, argv)  
int argc;  
char **argv;
```

In the main() function of the program, the user must specify all tuple space names with

```
LINDA_TS("tsname");
```

The One-in-all program must be started without the "-n <tsname>" option, of course! *Linda_main_ts()* is responsible to start all specified linda tuple space servers, except in the case the program was started with the "-s -" option.

And the simple init routine for the multi space needs no argument:

```
errstat  
linda_init_ts()
```

Linda type definitions:

```
typedef char      Linda_char;  
typedef char*    Linda_charp;  
typedef char*    Linda_string;  
typedef int      Linda_int;  
typedef int*     Linda_intp;  
typedef float    Linda_float;  
typedef float*   Linda_floatp;  
typedef double   Linda_double;  
typedef double*  Linda_doublep;  
typedef short    Linda_short;  
typedef short*   Linda_shortp;
```

The tuple mask argument is needed because the types of the following arguments can't be determined on runtime.

Tuple mask string

The tuple mask can contain the following identifiers:

1. Scalar actuals (data arguments in an *Out* or *In/Rd* operation)

Type	Identifier mask
Linda_char	%c
Linda_string	%s
Linda_int	%d
Linda_float	%f
Linda_double	%F

2. Aggregate (array) actuals (data arguments only in *Out* operation)

Type	Identifier mask
Linda_charp array (pointer)	*c:d
Linda_intp array (pointer)	*d:d
Linda_floatp array (pointer)	*f:d
Linda_doublep array (pointer)	*F:d

The array pointer argument must follow an integer value/variable with the size in type units, not in bytes!

3. Scalar formals (question fields)

Type	Identifier mask
------	-----------------

Linda_char	?%c
Linda_string	?%s
Linda_int	?%d
Linda_fbat (single prec.)	?%f
Linda_double (double prec.)	?%F

All formals arguments must be pointers of course.

4. Aggregate formals (arrays)

Type	Identifier mask
Linda_charp array (pointer)	?*c:d
Linda_intp array (pointer)	?*d:d
Linda_fbatp array (pointer)	?*f:d
Linda_doublep array (pointer)	?*F:d

The array pointer argument is followed by an integer with the size in type units.
Examples for tuple masks:

```
( "%s %d %d *F:d", "mystring", n, m, myarr, mysize )  
( "%d ?%d", 100, &i )  
( "%s ?%d %d ?*F:d", "mystring", &n, m, myarr, mysize )
```

Some constants:

```
TS_MAX_DIM      : maximal argument number ( 20 )
```

```
TS_MAX_THREADS : maximal thread number
                 using the linda interface
```

Warnings

Because the linda parser can only determine the tuple argument type from the tuple mask, you must carefully check the consistence of the tuple mask the argument list. If the parser expects a pointer, and you forgot the '&' operator, you will probably fail with a memory violation error!

Examples

Parallel calculation of the PI constant with random numbers.

1. Simple Linda Master/Worker modell

The master program master_pi

```
/*
 * Calculate the Constant PI with random numbers and the
 distributed
 * Linda parallel interface:
 *
 * In the x-y plane we construct a rectangular area with
 * side sizes of 2. We got a size of 1 (unit size) only for
 positive
 * (x,y) values.
 * We generate with a (high quality) radom generators positive
 values
 * in the range 0 .. 1 for each x- and y-coordinate and count
 * the numbers of generated random pairs with "allcalc":
 *
 *   (pos_x,pos_y)
 *   allcalc++
 *
 * We got a 2-dim vector with the length sqrt(rad):
 *
 *   rad=pos_x^2+pos_y^2
 *
 * Now we construct a circle of unit radius in the x-y-plane.
 * All points with rad<=1.0 are located within this circle. We
 * count these points with pi_hit.
 *
 * From these two counting variables we can calculate PI:
```

```
*
* PI=(pi_hit/allcalc)*4
*
* because the circle area  $A_{PI}=PI/4$  is 4 times
* smaller than the rectangular area. And the propability, that a
* random point hits in the cirle area is 1/4.
*
* Master control program
*
*/

#include "linda.h"

int
main(argc,argv)
int  argc;
char **argv;
{
    Linda_int  loopcount;
    Linda_int  maxcount;
    Linda_int  myid;
    Linda_int  finished;
    Linda_int  allcalc,pi_hit;

    double      pi_calc;

    int         maxclients;
    int         i;

    char  *linda;

    if(argc!=4)
    {
        printf("usage: master_pi <linda server name>
<maxclients> <maxcount>\n");
        exit(1);
    }
    linda=argv[1];
    if(!isdigit(*argv[2]) || !isdigit(*argv[3]))
    {
        printf("usage: master_pi <linda server name>
!<maxclients> !<maxcount>\n");
        exit(1);
    }

    /* maximal (x,y) point calculation of all workers */
    maxcount=atoi(argv[3]);

    loopcount=100000;
    maxclients=atoi(argv[2]);
```

```
printf("master_pi: maxcounts=%d maxclients=%d\n",
      maxcount,maxclients);

/* init the linda client interface */
linda_init(linda);

/* put the worker ids in the tuple space */
printf("master_pi: put myids\n");
for(myid=0;myid<maxclients;myid++)
    out("%s %d","myid",myid);

/* put the maxcount/loopcount values in the tuple space */
out("%s %d","maxcount",maxcount);
out("%s %d","loopcount",loopcount);

/* put an initial allcalc and pi_hit variable in the tuple
space */
out("%s %d","allcalc",allcalc);
out("%s %d","pi_hit",pi_hit);

/* wait untill all workers finished */
printf("master_pi: waiting for worker...\n");
for(finished=0;finished<maxclients;finished++)
{
    in("%s ?%d","finished",&myid);
    printf("master_pi: worker %d finished\n",myid);
};

/* collect the results */
in("%s ?%d","allcalc",&allcalc);
in("%s ?%d","pi_hit",&pi_hit);

/* and collcct our remains */
in("%s ?%d","maxcount",&maxcount);
in("%s ?%d","loopcount",&loopcount);

pi_calc=(pi_hit*1.0)/(allcalc*1.0)*4.0;

printf("master_pi: Result: allcalc=%d pi_hit=%d PI=%f\n",
      allcalc,pi_hit,pi_calc);

}
```

The client (worker) program worker_pi.

Must be started by hand (N times):

```
ax -m <host> worker_pi
```

```
#include "linda.h"

/* we use an own random generator version */

extern      Linda_double      rand_val01();
extern      void              rand_init01(Linda_int seed);
extern      Linda_int         rand_rand();

Linda_int   maxcount;
Linda_int   loopcount;

void hit();

int
main(argc,argv)
int   argc;
char  **argv;
{
    char      *linda;
    Linda_int  randinit;

    if(argc!=2)
    {
        printf("usage: worker_pi <linda server name>\n");
        exit(1);
    }
    linda=argv[1];

    /* init linda client interface */
    linda_init(linda);

    /* init random generator */
    randinit=-rand_rand()/1000;
    rand_init01(randinit);

    /* call the real worker */
    hit();

    return 0;
}

void
```

```
hit()
{
    int    mypass=0;
    int    indoor=0;
    int    myloop=0;
    int    i;
    Linda_int    allcalc=0;
    Linda_int    myid;
    Linda_int    pi_hit;

    Linda_double    pos_x,pos_y,rad;

    /* get the maxcount and the loopcount values from
     * the tuple space
     */
    rd("%s %d","maxcount",&maxcount);
    rd("%s %d","loopcount",&loopcount);

    /* get my id number */
    in("%s %d","myid",&myid);

    printf("worker_pi(%d): maxcount=%d loopcount=%d\n",
           myid,maxcount,loopcount);

    while(allcalc < maxcount)
    {
        mypass++;
        /* do the calucaltion loop */
        for(i=0;i<loopcount;i++)
        {
            myloop++;
            pos_x = rand_val01();
            pos_y = rand_val01();
            rad=(pos_x*pos_x)+(pos_y*pos_y);
            if(rad <= 1.0)
                indoor++;
        }
        /* get the allcalc variable value and add
         * our loopcount.
         */

        in("%s %d","allcalc",&allcalc);
        allcalc += loopcount;
        out("%s %d","allcalc",allcalc);

        printf("worker_pi(%d): Pass=%d\n",myid,mypass);
    }
}
```

```
/* okay, we're finished; add our indoor count to the
 * global variable pi_hit
 */

printf("worker_pi(%d): finished. indoor hits=%d\n",
      myid,indoor);

in("%s ?%d","pi_hit",&pi_hit);
pi_hit += indoor;
out("%s %d","pi_hit",pi_hit);

/* tell the master we're finished */
out("%s %d","finished",myid);

};
```

The one-in-all linda worker modell program example

```
pi_all -n "mypilinda" -N 4 -v \
      -r /profile/pools/linda \
      -- 100000000

#include "linda.h"

int
main(argc,argv)
int   argc;
char  **argv;
{

    LINDA_FUNC(worker);
    LINDA_MASTER(master);

    linda_main(argc,argv);

}

void
master(argc,argv)
int   argc;
char  **argv;
{

    Linda_int   loopcount;
    Linda_int   maxcount;
```

```
Linda_int    myid;
Linda_int    finished;
Linda_int    allcalc,pi_hit;
double       pi_calc;

int          i;

if(argc!=1)
{
    printf("%s: usage: %s {Linda options} <maximal
calcs>\n",
        Linda_my_name,Linda_my_name);
    return;
}

/* maximal (x,y) point calculation of all workers */
maxcount=atoi(argv[0]);

loopcount=maxcount/100;

printf("master_pi: maxcounts=%d maxclients=%d\n",
    maxcount,Linda_worker_num);

/* start the workers */
for(i=0;i<Linda_worker_num;i++)
    linda_spawn("worker",NULL);

/* put the maxcount/loopcount values in the tuple space */
out("%s %d","maxcount",maxcount);
out("%s %d","loopcount",loopcount);

/* put an initial allcalc and pi_hit variable in
 * the tuple space
 */

allcalc=0;
pi_hit=0;
out("%s %d","allcalc",allcalc);
out("%s %d","pi_hit",pi_hit);

/* wait untill all workers finished */
printf("master_pi: waiting for worker...\n");
for(finished=0;finished<Linda_worker_num;finished++)
{
    in("%s ?%d","finished",&myid);
    printf("master_pi: worker %d finished\n",myid);
};
```

```
/* collect the results */
in("%s ?%d", "allcalc", &allcalc);
in("%s ?%d", "pi_hit", &pi_hit);

/* and collcct our remains */
in("%s ?%d", "maxcount", &maxcount);
in("%s ?%d", "loopcount", &loopcount);

if(allcalc!=0)
    pi_calc=(pi_hit*1.0)/(allcalc*1.0)*4.0;

printf("master_pi: Result: allcalc=%d pi_hit=%d PI=%f\n",
    allcalc, pi_hit, pi_calc);

return;
}

/* we use our own random generator version */

extern    Linda_double    rand_val01();
extern    void            rand_init01(Linda_int seed);
extern    Linda_int       rand_rand();

Linda_int maxcount;
Linda_int loopcount;

void hit();

void
worker()
{
    char    *linda;
    Linda_int randinit;

    printf("worker %d started...\n", Linda_worker_id);

    /* init random generator */
    randinit=-rand_rand()/1000;
    rand_init01(randinit);

    /* call the real worker */
    hit();

    return;
}

void
```

```
hit()
{
    int    mypass=0;
    int    indoor=0;
    int    myloop=0;
    int    i;
    Linda_int    allcalc=0;
    Linda_int    myid;
    Linda_int    pi_hit;

    Linda_double    pos_x,pos_y,rad;

    /* get the maxcount and the loopcount values from
     * the tuple space
     */
    rd("%s ?%d", "maxcount", &maxcount);
    rd("%s ?%d", "loopcount", &loopcount);

    myid=Linda_worker_id;

    printf("worker_pi(%d): maxcount=%d loopcount=%d\n",
           myid,maxcount,loopcount);

    while(allcalc < maxcount)
    {
        mypass++;
        /* do the calucaltion loop */
        for(i=0;i<loopcount;i++)
        {
            myloop++;
            pos_x = rand_val01();
            pos_y = rand_val01();
            rad=(pos_x*pos_x)+(pos_y*pos_y);
            if(rad <= 1.0)
                indoor++;
        }
        /* get the allcalc variable value and add
         * our loopcount.
         */
        in("%s ?%d", "allcalc", &allcalc);
        allcalc += loopcount;
        out("%s %d", "allcalc", allcalc);

        printf("worker_pi(%d): Pass=%d\n", myid, mypass);
    }

    /* okay, we're finished; add our indoor count to the
```

```
    * global variable pi_hit
    */

    printf("worker_pi(%d): finished. indoor hits=%d\n",
           myid,indoor);

    in("%s ?%d","pi_hit",&pi_hit);
    pi_hit += indoor;
    out("%s %d","pi_hit",pi_hit);

    /* tell the master we're finished */
    out("%s %d","finished",myid);

    return;
};
```

3. Multi Tuple Space pi program

```
#include "linda.h"

#define PICALC "picalc"
#define RESULT "result"

int
main(argc,argv)
int argc;
char **argv;
{

    LINDA_TS(PICALC);
    LINDA_TS(RESULT);
    LINDA_FUNC(worker);
    LINDA_MASTER(master);

    linda_main_ts(argc,argv);

}

void
master(argc,argv)
int argc;
char **argv;
{
```

```
Linda_int    loopcount;
Linda_int    maxcount;
Linda_int    myid;
Linda_int    finished;
Linda_int    allcalc,pi_hit;
double       pi_calc;

int          i;

if(argc!=1)
{
    printf("%s: usage: %s {Linda options} <maximal
calcs>\n",
        Linda_my_name,Linda_my_name);
    return;
}

/* maximal (x,y) point calculation of all workers */
maxcount=atoi(argv[0]);

loopcount=maxcount/100;

printf("master_pi: maxcounts=%d maxclients=%d\n",
    maxcount,Linda_worker_num);

/* start the workers */
for(i=0;i<Linda_worker_num;i++)
    linda_spawn("worker",NULL);

/* put the maxcount/loopcount values in the tuple space */
out_ts(PICALC,"%s %d","maxcount",maxcount);
out_ts(PICALC,"%s %d","loopcount",loopcount);

/* put an initial allcalc and pi_hit variable in the tuple
space */
allcalc=0;
pi_hit=0;
out_ts(RESULT,"%s %d","allcalc",allcalc);
out_ts(RESULT,"%s %d","pi_hit",pi_hit);

/* wait until all workers finished */
printf("master_pi: waiting for worker...\n");
for(finished=0;finished<Linda_worker_num;finished++)
{
    in_ts(RESULT,"%s ?%d","finished",&myid);
    printf("master_pi: worker %d finished\n",myid);
};
```

```
/* collect the results */
in_ts(RESULT,"%s ?%d","allcalc",&allcalc);
in_ts(RESULT,"%s ?%d","pi_hit",&pi_hit);

/* and collcct our remains */
in_ts(PICALC,"%s ?%d","maxcount",&maxcount);
in_ts(PICALC,"%s ?%d","loopcount",&loopcount);

if(allcalc!=0)
    pi_calc=(pi_hit*1.0)/(allcalc*1.0)*4.0;

printf("master_pi: Result: allcalc=%d pi_hit=%d PI=%f\n",
    allcalc,pi_hit,pi_calc);

return;
}

/* we use an own random generator version */

extern Linda_double rand_val01();
extern void rand_init01(Linda_int seed);
extern Linda_int rand_rand();

Linda_int maxcount;
Linda_int loopcount;

void hit();

void
worker()
{
    char *linda;
    Linda_int randinit;

    printf("worker %d started...\n",Linda_worker_id);

    /* init random generator */
    randinit=-rand_rand()/1000;
    rand_init01(randinit);

    /* call the real worker */
    hit();

    return;
}

void
```

```
hit()
{
    int    mypass=0;
    int    indoor=0;
    int    myloop=0;
    int    i;
    Linda_int    allcalc=0;
    Linda_int    myid;
    Linda_int    pi_hit;

    Linda_double    pos_x,pos_y,rad;

    /* get the maxcount and the loopcount values from
     * the tuple space
     */
    rd_ts(PICALC,"%s ?%d","maxcount",&maxcount);
    rd_ts(PICALC,"%s ?%d","loopcount",&loopcount);

    myid=Linda_worker_id;

    printf("worker_pi(%d): maxcount=%d loopcount=%d\n",
           myid,maxcount,loopcount);

    while(allcalc < maxcount)
    {
        mypass++;
        /* do the calucaltion loop */
        for(i=0;i<loopcount;i++)
        {
            myloop++;
            pos_x = rand_val01();
            pos_y = rand_val01();
            rad=(pos_x*pos_x)+(pos_y*pos_y);
            if(rad <= 1.0)
                indoor++;
        }
        /* get the allcalc variable value and add
         * our loopcount.
         */
        in_ts(RESULT,"%s ?%d","allcalc",&allcalc);
        allcalc += loopcount;
        out_ts(RESULT,"%s %d","allcalc",allcalc);

        printf("worker_pi(%d): Pass=%d\n",myid,mypass);
    }

    /* okay, we're finished; add our indoor count to the
```

```
    * global variable pi_hit
    */

    printf("worker_pi(%d): finished. indoor hits=%d\n",
           myid, indoor);

    in_ts(RESULT, "%s ?%d", "pi_hit", &pi_hit);
    pi_hit += indoor;
    out_ts(RESULT, "%s %d", "pi_hit", pi_hit);

    /* tell the master we're finished */
    out_ts(RESULT, "%s %d", "finished", myid);

    return;
};
```

Literature

1. "p4-Linda: A Portable Implementation of Linda", R.M. Butler et al.
2. "Design and Implementation of a Tuple-Space Server for Java CSC 9020", A.B. Sudell
3. "Glenda Installation and Use", B.R. Seyfarth et al.
3. "Linda Tutorial", SCA (www.sca.com)

See also

`linda[U]` , **`linda_cap[U]`**

Tuple space server

Name

Linda - high performance Linda tuple space server for Amoeba's HP-Linda implementation

Synopsis

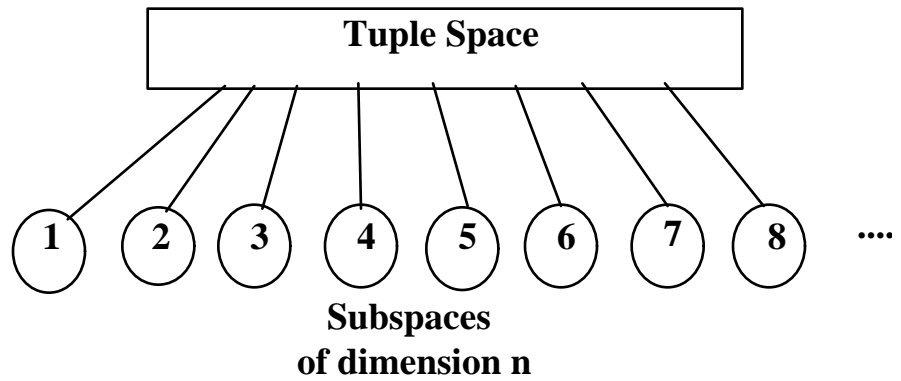
```
linda -n <tuple space name> [-v]
```

Description

The tuple space server *linda* is the central point of the design of the parallel extensions language Linda, developed first by the Yale University.

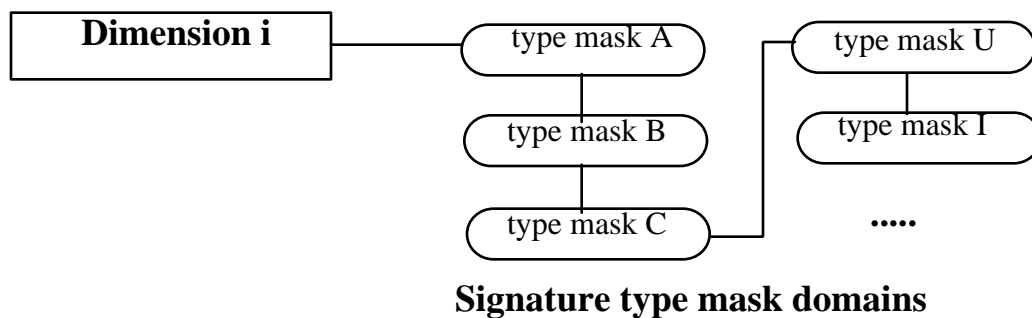
Fundamentals

The tuple space server service incoming tuple operation requests. The server must manage the various tuples in an efficient way. To achieve this goal, the tuple server divides first the tuple space in sub spaces, one for each dimension:



All tuples of the same dimension, that means argument number, belong to the same subspace N. The tuple server manage all subspaces with a static table with the maximal dimension number `MAX_TS_DIM`. IN and OUT tuples are handled in different tables and lists.

All tuple from a subspace are further divided in signature domains, that means all tuples with the same type signature, for example `[string,int,int,intp]` are merged in the same signature domain:



All subdomains of one specific dimension (argument number) are put together with a double linked list.

All tuples of such a sub domain with N arguments are combined with a double linked list again.

Searching

To find a matching tuple (for example an IN operation with a template tuple T), the tuple server first determine the tuple dimension and the type signature mask. It then searches the specific list of all sub signature domains for the matching type mask. If it was found, the tuple server must compare the arguments of each tuple in this subdomain with the template T .

To speed up this search, a byte checksum for each scalar tuple argument was previously calculated. From the template tuple, too. The byte checksum are packed in a long array. Therefore, to do a first matching decision of the first 4 tuple arguments, only one long compare is needed. If this compare fails, the next tuple can be searched. Only scalar arguments will be compared. Formals (question fields in the template tuple) must be masked out before the compare of the signature checksum array of the tuples and the template. If the first decision compare is true, all argument will be compared in respect to their data types.

To avoid specification limitations in the tuple server, no hash search was used. In contrast to other public Linda implementation, hp-linda needs no specific keys in the tuple argument list. Therefore, the first argument can be a formal, or one of the supported Linda data types.

Client-Server Communication

The server port will be created from a user chosen tuple space name. The public server capability will not be installed in the filesystem. The client can create the server port if it knows the tuple space name, too. To avoid server abuse, the private field of the server and public capability will be taken from the users root capability. Only the user itself has the right

one.

Options

-n "tsname"

To make client and server communication to be possible, the tuple space server needs a tuple space name. From this name, the server port and the public port will be derived. To avoid abuse from the outside, the private part of the server and public capability is retrieved from the users root capability private field.

-v

Verbose mode. Gives more information.

Administration

Special administration is not needed. Usually, the linda server will be automatically started and terminated by the linda user program. But it's possible that the linda user (master) process dies on a fatal error. This leads to a ghost linda tuple space server. To exit the linda server in a clean way, use **lindacap[U]** to create the tuple space server capability in the filesystem. You can use the **std_exit[A]** tool to finish the linda server.

```
lindacap -n "mytuple space" -c "tscap"  
std_exit tscap
```

Examples

```
linda -n "mytuple space"
```

See also

linda[L] , **lindacap[U]**