Message Passing and MPI

CS-594

Understanding Parallel Architectures: From Theory To Practice

Dr Graham E. Fagg

Overview

- Covers message passing and the MPI standard and API
  - history of message passing
  - Covers more advanced features of MP systems and MPI and why they exist
  - Examines some of the current issues in using these systems and what the future holds
    - multi-protocol systems, intro to metacomputing, scheduling and cluster computing issues

Lectures 2

- Lecture 2 Message Passing
  - What is Message Passing
  - History of Message Passing
  - Overview of MPI
    - Including some numeric program examples
  - How do we use MPI
    - From compiling to fixing mistakes and getting it really running

Getting help!

- Contact the TA first
  - TA: invisible man
- Contacting me:
  - Email me first: fagg@cs.utk.edu
  - 320 Claxton, Phone 4-5790
  - Best time to find me late afternoons and evenings

Message Passing

- What is message passing?
  - Its where tasks or processes communicate via explicit send and receive operations on fixed items of data…
  - As opposed to?
    - Shared memory where normal read/write operations can be used to ***SHARE*** data.

Message Passing

- Data can be passed between processes on the same machine or between processes on different machines.
- They might not even exist at the same time
  - I.e. no need for temporal coupling but this is unusual

Message Passing

- Why message pass?
  - So we can break problems into smaller pieces for faster performance (DMMP / MIMD)
  - For fault tolerance (multiple servers)
  - Explicit operations can be reasoned about in a formal way, I.e. CSP
  - Message Passing can be standardized to allow for highly portable applications
    - Was not always true
    - Is now happening to shared memory, see OpenMP

Message Passing

- Many computer systems cannot share resources in a transparent way.
- Even we can, we can implement message passing on top of that shared resource.
  - This sometimes yield great performance increases
- Message passing allows systems that avoid contention on resources
  - Compare the IBM SP-2 to the SGI Origin2000.

Message Passing

- How long has it been around?
  - Since before networks!
    - Copy data (known as a message onto a removable media, such a a removable disk)
    - Pass the message (I.e move the disk across the room/building/country)
    - Read the message (mount the disk and read it)
  - Not that silly an example, until recently some universities in NYcity passed data via motorbike couriers on Exbyte tapes
  - Until ATM has finally caught up in BandWidth..

Basics

- We need two or more entities
  - A sender
  - A receiver or receivers
- Some data, the *message*
- Some means of passing the data
  - network, shared resource…
- The two basic operations
  - Send and Receive
- And a means of identifying each entity, and the message… *maybe*

Basics

- The Sender passed the message by:
  - sending it:
    - send (data, somewhere, args)
  - The receiver gets it by asking for it:
  - receiving the message:
    - receive (some buffer, some other args)
- The variations will be examined in a few slide time.

The history of message passing as we know of it

- ARPA net users passed message via protocols such as IP
- E-mail is a message passing system..
- But, we are interested in the ‘true’ parallel computing versions..

Past message passing systems

- How the facilities and functionality of modern message passing systems evolved has been influenced by a vast number of research projects and commercial implementations by vendors of MPP machines.
- We will briefly discuss some of these systems in terms of which features (we now think of as standard) that they introduced and what they omitted. This will also help you understand how implementers of such systems have learned to insulate users more effectively from the increasing complexity of the underlying systems.

Vendor Message Passing Systems and Machines

- First we will cover the hardware systems and their message passing systems that led to today's range of systems
- Then we will cover some portable message passing systems
- Not an exhaustive list, but long enough to show just how varied it can be.

Caltech Hypercube

- The Caltech Hypercube (circa 1984) was a d-dimensional hypercube structured system with a computational node at the end of each vertex, and a single host to control the machine (known as an Intermediate Host).
- The system was programmed in either C or Fortran77 and communication was based on a subroutine library known as the Crystalline Operating System (CROS).
- The communications library assigned addresses to tasks depending on which node they were physically located, processes could only communicate to neighbors or the intermediate host (a total of d+1 links). The CROS terminology for a link between two nodes was that of a channel though which 8 byte message packets could be sent.

Caltech Hypercube

- The system only supported collective operations (broadcast) to/from the intermediate host and the overall communication pattern was SIMD in nature.
- I.e. all processes had to call the same communications routine at the same time. This lead the machine to appear halfway between the earlier SIMD machines such as the ICL DAP and Thinking Machines CM-1/2 and the later Intel iPSC and XPS Paragon machines. The former where program execution and communication occurs fully in lockstep and the latter where ordering was completely independent.
- On the Hypercube under CROS, the program was free to run independently to each other but the hardware forced all the communication into lockstep.
- For solving very regular problems in physics such as partial derivatives for large numbers of grid points, the structure imposed by the programming environment was an aid for producing low-level very efficient implementations.

Caltech Hypercube

- This was a ``difficult target for programming any but the most highly regular problems".
- This led to the development of the "Distributed Process" environment a kernel based small operating system which allowed greater control and flexibility than CROS. In particular, the user developing message passing programs no longer had to think about a fixed mapping between nodes (running programs) that communicate in a fixed way via channels (i.e. hardware links) but could now think about processes connected via virtual communication channels
  - the basic address independence from hardware location abstraction that is now common place.

Caltech Hypercube

- Went from a fixed broadcast (I.e. no addresses specified in the send operation) to an addressed based system.
- Making everybody do a send and then receive the same time… not nice.
- Note the small message size of 8 bytes
  - If you wanted more, you had to write your own message passing layer… nasty.

Meiko CS-1 and Occam

- Transputer based multiprocessor
- The transputer was a 32 bit microprocessor that had communication hardware built in.
  - 4 high speed links
- The transputer could context switch in a single cycle
  - I.e. multitask-mutlithread very quickly
    - more than one process per processor

Meiko CS-1 and Occam

- Occam was a language based on the CSP specification language
  - CSP - Communicating Sequential Processes
  - CSP could be formally reasoned about
    - Popular target for S/W Eng projects in the early days of ESPRIT. Many CS-1 machines where supplied under the ESPRIT ALPHA Project
- Occam was a parallel language

Occam

- Instructions were executed in order of blocks
  - blocks could be executed internally as parallel (par), sequential (seq) or alternately (alt) which meant no-deterministically
  - Seq
    - do A
    - do B
  - A happens before B always

Occam

  - Par
    - do A
    - do B
  - I.e. they execute together or after one another

  - alt
    - do A
    - do B
  - We don't know which one executes in which order.

Occam and message passing

- Message passing was built in
- Based on channels between processes
  - a channel is a bi-direction pipe much like unix sockets
- Sending a message
  - chan1 ! data
- Receiving a message
  - chan1 ? buffer

Occam and message passing

- Message passing was synchronous and blocking
  - I.e. both parties had to work together to message pass, and the send and receiver would wait until their operation had completed completely before the program would continue
- Was easy for programmers to write code that deadlocked
  - lucky we had lots of ESPRIT analysis tools to help

Occam and message passing

- Deadlocking code
  - Proc1
    - chan1 ! Data
    - chan1 ? Buffer
  - Proc 2
    - chan1 ! Data
    - chan1 ? Buffer
- This problem happens in many systems without the MP system buffering for you.

Occam lives on

- The Occam project still lives on where formally proved code is needed.
- Newer versions allow recursion which was only possible by loop contracts previously (see Occam 2 1/2)
- Compilers are available for systems other than transputers such as Sparc, PowerPC etc
  - Kent retargetable Occam compiler project {Welsh96} from the University of Kent at Canterbury.
  - SPOC - The Southampton Portable Occam Compiler from ECS, University of Southampton.

NX and the Intel iPSC1 and the Intel Paragon

- The original iPSC1 (circa 86) was a seven-dimensional hypercube structured system much like the Caltech Hypercube in terms of hardware design although it software environment known as NX1 was less like CROS and more like the ``Distributed Processes environment.
  - iPSC? The Intel Personal Super Computer for those who could afford a personal supercomputer….
  - The comp.parallel Usenet news page was originally aimed at iPSC users.
  - One of the largest Paragons XPS machines ever installed is at ORNL
- The system was very well balanced, I.e. its computational power was proportional to its message passing performance.
  - Far quicker than even more modern systems like IBM SP2s (like the new ASCI Blue Pacific machine) or even many Cray T3D/E machines.

NX and the Intel iPSC1 and the Intel Paragon

- The NX1 operating system was based on the Caltech Reactive Kernel which provided hiding of the underlying communication topology (processes where identified by a simple integer from 0 to P-1, where P was the number of processes per partition), multiple processes per node, and any to any message passing, non-synchronous messaging.
  - i.e. both sender and receiver do not have to be active at the same time for communication to complete) and non-blocking (i.e. no need to wait for completion)
- what is now thought of as a typical set of features that define a message passing environment.

NX and the Intel iPSC1 and the Intel Paragon

- This additional flexibility also added an increase in the complexity of the message passing library. Users now needed additional routines to inquire location information, and messages needed to be addressed correctly
  - Additional arguments in the subroutine calls.
- Messages needed to be identified on an individual bases as there was neither a fixed order of transmission and therefore receipt, but the pattern of communications was no longer dictated by topology.
- To assist developers, messages could be tagged or typed.
  - Like a subject like in E-mail!
  - a user assigned integer could be associated with a message which would be used by the receiver to distinguish messages.
    - Unfortunately the original system did not permit the filtering of messages by sender identity and type at the same time, although the type could be set to the senders ID to allow for receive from sender semantics.

NX and the Intel iPSC1 and the Intel Paragon

- An additional new feature was that different length messages could be received prior to the receiving process knowing which was which and hence knowing how big a buffer memory to allocate, or even how much data was received.
- The user had to indicate for each message how large the receive buffer was. Only after the receive completed could the user find out how much data was received, up to the maximum allowed of 16Kbytes
  - a vast improvement over the Caltechs Hypercube 8 bytes
- If the user offered a buffer that was too small the the excess message data was discarded (truncated).

NX and the Intel iPSC1 and the Intel Paragon

- The later Intel machines implemented an improved version of NX, known as NX2. In the case of the Paragon, this was implemented upon an OSF/1 micro Unix kernel.
- A number of improvements were added such as interrupt driven communication which allowed an application to perform computation and be woken up when a message arrived instead of having to poll for them intermittently (leading to decreased cache performance and possible page faults as the OS calls are invoked to check for messages).
- I.e. Asynchronous and non-blocking messaging
  - Be careful many users call non-blocking asynchronous and vice versa

NX and the Intel iPSC1 and the Intel Paragon

- Other changes included the inclusion of message identifiers (mids) that allowed simple identification of non-blocking operations. When a non blocking operation is initiated, a mid would be issued and the user could check for completion of this mid later, thus allowing the underlying hardware communications processors to overlap communication while the compute processors continued.
- Semantic changes included allowing the use of wild cards (typically negative integers) to denoted groups or processes.
  - receiving a message of type -1 would denote receive from anybody of any type, i.e. whatever was received next.
  - Sending to a node of type -1 would denote sending to all processes on a partition.

NX and the Intel iPSC1 and the Intel Paragon

- Up to this point communication had been point to point, i.e. from a single sender to a single receiver. New broadcast functions allowed the construction of global operations such as global synchronizations (barriers) and some arithmetic reduction operations.

NX and the Intel iPSC1 and the Intel Paragon

- Although NXs design inspired many of the features of current message passing systems, it also had a number of short comings:
  - such as lack of more comprehensive group communication functions (to assist certain types of calculations)
  - lack of message identification and filtering at the receivers end
    - only one type compared to up to three used on later systems, and
  - initially a high software overhead compared to simpler protocols such as active messages which had direct access to hardware.
- But, it did give us the semantics of the most common message passing systems in current use

IBM Scalable Power series and EUI

- The IBM Scalable Power (SP) Series of systems starting with the IBM 9076 SP1 and the later SP2 machines consisted of tightly coupled distributed memory set of RS/6000 RISC processors interconnected by a high speed switch.
- The systems were based on experience gained from the Vulcan hardware project and the Viper operating environment.
- The system designed to program these systems were known as the IBM external user interface (EUI) and consisted of four main components:
  - task management, message passing (point-to-point), task groups and collective operations.
- The last two items being of particular importance to later systems such as MPI.

IBM Scalable Power series and EUI

- Point to point communication under EUI was performed by sending messages to tasks directly in the same style as on the Intel iPSC systems
  - addresses from 0 to N-1, where N was the number of tasks making up a parallel job.
- The point to point system supported typed messages for both blocking and non-blocking messages.
  - Unlike NX, messages could be selected by the receiver on both message type and source (sender) including the use of wild cards.
- To assist in handling non-blocking messages the user could check the status of a particular transfer as well as wait for completion of a named operation, any of a range of operations or all pending outstanding transfers.

IBM Scalable Power series and EUI

- EUI allowed the construction of conceptual collections of tasks into logical groups that could be addressed by a single group ID.
  - See PVM groups latter.
- This allowed users to avoid having to list (sometimes large numbers of) tasks explicitly when passing messages in structured ways repeatedly.
  - The use of collective operations on these groups avoided the use of many point to point calls and allowed the system to perform these as efficiently as possible on the given hardware.
- The range of operations included, barriers, data shifts, broadcasts, gather, scatter, a generalized combine and an associative reduction.
- All the collective operations required all members of each group to partake in a blocking fashion.
  - The term 'collective' used in MPI comes from the IBM system.

IBM Scalable Power series and EUI

- Asynchronous returns from non-blocking functions required a two part status lookup. If the user interface to a non-blocking receive was as follows:
  - `void recvf (&data, sizeofbuffer, &size-received)`
- The size-received variable could not be filled in by the system until the non-blocking operation had completed, which might be while the thread that made this call was in a different program module. Thus the need to get a status handle which could be queried after the operation had completed and who's memory storage was handled by the messaging system.
  - `recvf (&data, ... &status)`
  - `...`
  - `wait (for above recv to complete)`
  - <- status is now safe to examine, as the wait operation has completed any status data structures.

IBM Scalable Power series and EUI

- The EUI project was not the first to introduce this two step strategy, this also occurred between NX1 and NX2, but the overall design used by IBM was the bases of that used by the MPI forum.
- Also much of the EUI collective operations design also directly effected the design of MPI.

Meiko CS-2 and CSTools

- The Meiko CS-2 grow out of many of the lessons learned from the SUPRENUM project, especially in terms of the Meikos ELAN message handling hardware coprocessor and the coupling of a compute engine (i860 and later Sparc based processors) with optional vector floating point units at each node.
- Each node ran a copy of the Solaris Unix micro kernel which mapped the Elan memory map into User space and thus provided access directly to the communications hardware without having to invoke a system call (protected mode) which drastically increased performance.
  - Also meant that each node was single user only as much of the speed improvement came from not having to check user Ids and permissions.
- This overcame the shortcoming of previous designs which had suffered from poor performance due to software overheads in accessing hardware like the NX library running under OSF/1 on the Intel Paragon for example.

Meiko CS-2 and CSTools

- The system was programmed with the previously developed CSTools environment which used the concept of named communication channels known as transports through which messages could be passed.
- Hence users could code in terms of these named links rather than in terms of the actual end-points.
- Once a transport had been instantiated repeated communication through it incurred very little system overheads.
  - A feature later used in MPI (known as persistent communications)
- Blocking, non-blocking, synchronous and asynchronous point to point operations were all supported.
- Largest installed machine was at LLNL
  - Which required the company to move its head office from Bristol, England to CA.

Thinking Machines CM5 and the CMMD Active Message Layer (AML)

- The Thinking Machines Connection Machine 5 (TMC CM5) was very different from previous Connection machine designs, being a true distributed memory MIMD system as opposed to the previous SIMD systems.
- The machine featured two interconnection networks, and Sparc based processing nodes each with four vector units for pipelined arithmetic operations.
  - Sounds a bit like the Meiko CS-2….
- The programming environment consisted of the CMOST operating system, the CMMD message passing system and various array style compilers the most popular of which was CMF a F90 style SIMD programming system.

Thinking Machines CM5 and the CMMD Active Message Layer (AML)

- The CMMD message passing system was unique in that it offered users access to routines from the lowest level, Active Message Layer (AML), point to point, channels and a cooperative functions library.
  - The AML system was the lowest level of operation and manipulated the communications hardware directly. The layer provided three basic operations:
    - active messages, which are like a lightweight RPC call. This is where the sender sends the address of a function to be invoked at a remote node together with its arguments in a single 72 byte packet.
    - The second operation was that of a data transport mechanism which only wrote data into a remote memory at a set location (much like Crays shmput operations).
    - The final operation was a receive port data structure which assisted in handling multiple data packets.

Thinking Machines CM5 and the CMMD Active Message Layer (AML)

- The CMMD point to point library was built on top of the AML, and provided the common list of operations including:
  - blocking, non-blocking, synchronous and asynchronous point to point operations with selection upon source, message type or wild cards.
  - Interestingly the blocking calls were quicker than the non-blocking calls as they avoided system level copying of message data.
- Another interesting point was the inclusion of a "send and then receive" operation.
  - This allowed for simpler coding of stencil operations and boundary exchanges in domain decomposition problems.
  - The system implementation of the joint send and receive operation being quicker than the two separate operations.
- The send-receive operation was so useful that it was also provided for in MPI.

Thinking Machines CM5 and the CMMD Active Message Layer (AML)

- For example previously to exchange values two operations would be required:

```
– Task A            Task B
– Send (B, data)    make copy of data edge into data'
– Recv (B, data)    Recv (A, data)
–                   Send (A, data')
```

- As opposed to:

```
– Task A        CMMD_send_and_receive (B, data)
– Task B        CMMD_send_and_receive (A, data)
```

- Note the two operations version is made more complex by the need to copy data values to prevent them being over written before they are copied into the message layer, a common complication found in many wavefront calculations.

Thinking Machines CM5 and the CMMD Active Message Layer (AML)

- CMMD Virtual Channels
  - Like the transport operations on the Meiko CS-2, these channels would allow multiple communications to occur with minimal overhead once they had been initiated.
  - The interface was very basic with calls to open, close and check the status of channels. A write operation was provided, although the receiver would have to check status to find out if its data had arrived, and then reset the channel to allow for more data to be sent.
- CMMD Cooperative Communications
  - These included broadcast, reduce, synchronize (barrier) and scan functions. Unlike other systems these used a separate interconnection network to provide performance.
    - Another recent architecture to also use separate hardware to enhance performance of global operations is the SGI Cray Research T3D/E machines which utilizes a special signaling line for handling barrier synchronization.

German Suprenum Project

- German project to create a made in Germany SuperComputer, with German made compilers and parallel computer languages and tools.
- For a time were the fastest machines in the world at 5 Gflops.
- Benchmarking the machine led to the follow up ESPRIT Genesis project.

German Suprenum Project

- Two separate systems were used to program the machine:
  - one was the PEACE operating system which through a complex compiler allowed a mix of message passing and Fortran 90 array extensions to each program running on each node.
  - The other was a version of PARMACS which is discussed later in more detail.
- Above the PARMACS system, high level libraries such as GMDs COMLIB, a grid based tool were created to assist users in using problem domain specific numeric solvers.
- Other examples included LiSS a parallel multi-grid solver for partial differential equations.

German Suprenum Project

- The use of PARMACS is important, as the software designers not only concentrated on improving the range of group operations available to users but also realized that portability of code was important especially in the light of the short life span of each MPP.
- Note on short life span.
  - Although the Suprenum project was very successful in some research areas such as compilers, and interconnections (which led to Meikos ELAN system) only 5 machines were delivered
  - The system suffered from the TRAP 71 error, where CPU and communication boards would just fail and damage themselves randomly!
  - The user would receive the *Helpful* error message "OS error: Trap 71"
    - The German idea of high availability servers?

So what did we get?

- From a very restrictive system (Caltech Hypercube) to systems with multiple ways of sending a message that is addressed in a flexible way and tagged.
- Receivers have multiple ways of filtering messages (using addresses, tags, channels) and can start a receive and get back to it when finished.

m4 and p4 macros

- P4 was a system, that grow out of a set of fortran macros that was was developed at Argonne National Laboratory (ANL) for use on a HEP shared memory computer.
- The original Macro's were called MonMacs that were processed at compile time by the Unix m4 preprocessing utility, and offered the user a set of monitor functions used to provide locks on critical sections of code that accessed shared data.
- The use of macros avoided an additional set of stack operations if the monitors had be based on function calls.
- The authors of the system co-wrote the book ``Parallel Programs for Parallel Processors" which give the system its final name of p4.

m4 and p4 macros

- The system was used as a bases for several specialized versions
  - TCGMSG for Chemistry problems
  - GMD macros for solving problems on regular grids
  - GMD macros was the basis for the very successful PARMACS system developed by Rolf Hempel at GMD in Germany.
    - Used originally on the Suprenum machines.
    - He is currently at NEC, Germany. And NEC has one of the fastest MPI implementations in existence for their SX series computers.

m4 and p4 macros

- The final p4 system was based on procedure calls and supported C as well as Fortran on a very wide range of systems including both distributed memory as well as shared memory systems.
- The programming paradigm was that of procs (processes) that formed administrative clusters, that intercommunicated by either locks or explicit message passing.
- The system provided user access to buffers so that they could avoid additional buffering by the system if they were knowledgeable enough.
- There was no non-locally blocking or asynchronous calls. I.e. the calls returned when the data was sent, and the user did not have to probe, test or wait for completion before reusing buffers.

m4 and p4 macros

- One globally blocking point to point call was also included p4sendr() which waited for an explicit acknowledgment from the receiver before returning, hence the **r** on the end of the call name to signify a rendezvous.
- The system also included a wide range of collective operations, with the ability to use the p4_global_op() call to construct user defined operations.
- Although the p4 system was very efficient, it was not as popular as other message passing system and was later used as a layering scheme to support other projects such as:
  - Chameleon {Gropp93}
  - BlockComm
  - MPI in the form of MPICH {Gropp94,Gropp96}. As you will USE.

The Express Environment

- Express {Parasoft90} was initially based on the Crystalline Operating Systems message passing library from Caltech.
- The initial implementations emphasized performance and offered good performance, with specially tuned versions of the software for certain MPP systems, which also supported limited dynamic process creation.
- The system later grew into a complete application development package which supported dynamic load balancing, parallel IO and comprehensive collective libraries for common topologies such as rings, grids and tori etc.

The Express Environment

- These topology features were made accessible through the exgrid*() family of calls.
- If used correctly a complete message passing application could be created by using exlayout() to specify distribute data and exdist() to move the data, leaving the user with no explicit send and receive calls, thus avoiding mis-ordered calls and complex handling of boundary conditions when implementing numeric solvers.
- Bottom line
- It was fast, did most of the work for you, was a complete programming environment.
- Was popular in Europe, even though it was expensive to buy.

Zipcode

- Zipcode{Skjellum94} was an experimental portable message passing system that emphasized support for parallel library development.
- As most message passing systems allowed the use of wild cards during reception of messages, it made correctness of message passing libraries and message passing user programs together very difficult as there was no guaranteed means by which to keep their messages from getting intermixed.
  - Zipcode introduced safe communication spaces, so that separate program units (libraries) could have their own space within which to communicate.
- This was achieved by making the addressing of processes relative to static process groups combined with a message context, a system supplied additional message tag, that was opaque to the user.
  - The process groups (with relative addresses known as ranks) were bound to contexts by processes known as mailers.

Zipcode

- Zipcode also allowed for topology information to be associated with process group information so that communication addresses could be specified relative to current position in topological terms
  - i.e. send message to next in ring, rather than having to look up an absolute address and then specifying that value as an address in the send operation.
- Much of the topology information and context handling went into the design of MPI.

Linda

- Linda{Carriero94} is based on an associative shared virtual memory system, or Tuple Space.
- Rather than sending messages from one process to another, processes create, consume, duplicate or evaluate data objects known as tuples.
- Creating a tuple is performed by calling out(), which is then passed into a common shared space which all other processes have access to.
- Reading a tuple is performed by either rd() or in(), where rd() just reads (duplicates) the tuple and in() consumes the tuple and removes it from the shared space.
  - Matching of tuples is performed by specifying a template of possible fields in the tuple when calling either rd() or in().

Linda

- eval() creates tuples asynchronously, with each field making the tuple evaluated in parallel (or at least in a non-deterministic order much like the Occam ALT construct).

Linda

- Linda allows for very simple but powerful parallel computer programs by simplifying addressing
  - no explicit addressing of processes, only the data they handle
  - and by removing the coupling between tasks. In Linda, you cannot specify the consumer of a tuple, and the consumer might not even exist when the tuple is created.

Linda

- For example in a conventional message passing system to pass a message from one process to another:

```
– task A
– find address of 'B' (addrB)
– Send (outdata, addrB, messagetag)

– task B
– find address of 'A' (addrA)
– Recv (indata, addrA, messagetag)
```


Linda

- Where in Linda we could have:

```
– task A
– out (messagetag, outdata)
– exit
```

- some time later,

```
– task B
– in (messagetag, ?indata)
– {process data}
– exit
```


Linda

- In this case the tuple that contained 'messagetag' as the first field would be consumed by task B (the '?' specifies a wild card) upon which its data would be placed in the indata memory buffer.
- Neither process ever having overlapped temporarily or having known of each others 'address'.

Linda

- Although initial implementations were slower than native libraries, later versions that utilized compile time analysis of data fields used by application programs allowed the run-time system to select tuned low level services that implemented the tuple space management and matching operations.
- On a number of tests{Deshpande92} some versions of Linda performed comparatively to both networked message passing systems such as PVM as well as native vendor message passing libraries on a number of MPPs for medium to large message payloads.

What have we got this far?

- Non-vendor versions are more portable
- Some are more efficient than others
  - some are almost as fast as the vendor systems
- The number of features has increased
  - group communications better supported
  - first portable high level libraries appearing
  - layering approach appearing
    - MPI based on the CH ADI on top of p4…. Or even PVM

Whats next after the break

- MPI standard and API
  - You will need to look at those handouts!
  - What we will cover
    - Safe communications – communicators, groups
    - Message passing semantics – blocking, non blocking, local, global operations
    - Collectives
    - Buffering and data types
    - How to compile and run MPICH
    - Some simple examples
    - The homework