Presentation 9

Message Passing Architecture

מודל העברת הודעות הוא צורת העבודה במחשבים גדולים ומחשבי על. המודל הזה מתאים למקבילות ברמת הפתיל (TLP) וגם ברמת התהליך, ובעיקר בעיות גדולות שמבצעים חישובים רבים בין החלפת תוצאות בין פתילים. עם אפשרות לחלק בעיות למיליוני פתילים במקביל, העבודה בדרך כלל מחלקת קבוצה גדולה מאוד של נתונים לתת-קבוצות ומקדישה מעבד שיעבוד על הקבוצה. מניעת סיכונים (data hazards) נעשית ברמת ההודעות שמזהות את המקור והגיל של כל תוצאה.

Slide 2 – 4 **Message Passing Multiprocessor**

In **message passing**, the system is made up of N nodes (0 to N – 1), each node consisting of a CPU with its private address space A (0 to A –1). CPUs cannot generally access the memory of any other node. Threads running on the N nodes communicate among themselves by formulating, passing, and reading inter-CPU messages over the internal network. The messages combine data and memory synchronization information. Although multicore processors in a node share memory locations, the primary decomposition of work in parallel algorithms is performed by message passing.

Interprocessor communication involves transactions for sending and receiving messages defined in the **Message Passing Interface** (MPI), an API for message passing parallel programming. The API defines transaction primitives between the processor and the interprocessor network — the implementation of these primitives depends on the OS and the hardware. The interprocessor network is understood as a general computer network, whose performance is determined by the communication protocols and the efficiency of its message types, routing, network access, message segmentation/reassembly (SAR). For this reason, with adequate network capacity, message passing is scalable to large systems with no geographical restriction, and grows naturally into cluster computing. In many current implementations, MPI is mapped onto the TCP/IP protocol stack in the OS.

The basic message types perform: **send data**, **send request** (for data), **send response** (to a data or status request), and **send system command**. The messages are structured, including as **header** (source ID, destination ID, timestamp) and **content** (data / request / response). On the one hand, the requirements of message management are the main source of processing overhead. On the other hand, **sequential consistency** is enforced in message passing without shared memory or snooping overhead. The code expects specific content from a specific source, so that the message header provides a consistent write order.

Slide 5 **Message Passing Example — Scalar Product**

Slide 5 shows data decomposition in a message passing system:

1. Each node performs one multiplication,
2. P0 and P2 send their result to P1 and P3 that are expecting to receive them and perform partial sums,
3. P1 sends its new result to P3 that is expecting it and performs the final sum

P3 receives two messages in its code. There is no data hazard (such as out-of-order message arrivals) because the data is labeled by the source ID in the message header.

Slide 6 Blocking versus Non-Blocking Models

Every program must pair a send instruction in one thread with a receive instruction in another thread. In **synchronous** (blocking) message passing, threads synchronize on the send/receive pair. Send and receive instructions block in code — the processor cannot proceed to other instructions until the message transmission is completed. To create the synchronization barrier, the receiver waits for an RTS (request to send) signal and the sender waits for a CTS (clear to send) signal.

In **asynchronous** (non-blocking) message passing, the sender executes the send instruction and continues processing new code. The message is buffered at the sender until being sent through network, and it is buffered at the receiver until it is needed. The receiver accepts the message by executing a receive instruction. This method can be compared to the operation of TCP in internet connections.

Slide 7 – 11 MPI Messages

Slide 7 shows some MPI environment messages. The directives are sent by the application code to the OS to initialize, configure, and close the MPI environment.

Slide 8 shows some MPI point-to-point messages, that is, directives for sending messages from one node to one other node. These include the blocking and non-blocking forms of basic send and receive directives.

Slide 9 shows some MPI collective communication messages, that is, directives from one node to a collection of nodes. These include barrier synchronization, broadcast to a group of nodes, and the collective messages **scatter**, **gather**, and **reduce**. The scatter and gather operations are shown on slide 10. In scatter, one node distributes a list of data sets to a list of destination nodes (usually each node receives one data set). In gather, one node receives and assembles a list of data sets from a list of source nodes (usually each node sends one data set).

Slide 11 shows the reduce directive. It is similar to gather, except that the OS performs a reduction operation on the received data sets, such as adding the list of results assembled from the list of sources.

Slide 12 – 14 MPI "Hello World"

Slide 12 shows a simple program that sends the "Hello, there" string from the master thread (process 0) to thread 1. Thread 1 receives and displays the message. Additional details of this use of the send and receive directives are given on slide 13. Slide 14 shows the use of scatter and reduce to implement the scalar product in MPI.

Slide 15 – 17 Implementation Issues for Processor

The requirements for a processor directly supporting message passing include:

1. A physical send and receive port for message transmission
2. Send and receive message buffers
3. Hardware blocking on message access for synchronous messages

4. Interconnect support for message passing

If there is no direct support for message passing in the processor, then the OS and interconnect network must buffer messages for the processor and handle synchronization.

Slide 16 shows the plan for the Alpha 21364 CPU designed to support message passing. The processor package was designed around the Alpha 21264 RISC core with L1 cache. The Cbox in the center of the package is a communications controller connecting the main memory buffers (Z0 and Z1) designed for Rambus (an early version of fast DDR RAM), the System Data Buffers (for the message passing system), and an internal router that connects the package to similar CPU packages and to peripheral I/O. Two configurations for multiprocessor systems built from the Alpha 21364 are shown on slide 17. The connections between any two Alpha packages are made to the N, E, W, S connections to the internal router. In its time, the Alpha 21264 RISC core was the fastest processor on the SPEC list. The 21364 was never manufactured for commercial reasons.

Slide 18 – 19 Cluster Computing

Cluster computing refers to a large message passing system with distributed memory. It can exploit MPI scalability up to millions of nodes. The typical node is a standard workstation. The node-to-node scale may be up to a few meters (interconnected through a physical bus or non-blocking crossbar switch), a few hundred meters (interconnected through a LAN), or up to thousands of kilometers (a WAN). When the interconnection network is a LAN or WAN, the MPI_COMM_WORLD list includes MAC addresses or IP network addresses. Cluster computing enjoys the advantages of large message passing systems, as reviewed on slide 19.

Slide 20 MPI and Socket API

The socket API maps each endpoint (IP address and port) to a socket number. The socket number identifies a connection to a remote endpoint. At the client, the socket identifies a connection to server endpoint, and at the server, the socket identifies a connection from a client endpoint.

In systems using both the MPI and socket APIs, joining an MPI cluster can be implemented as a client/server system. The primary node acts as a client and opens a connection to each remote server. The secondary nodes run an MPI server listening on a well-known service port number. MPI maps the socket numbers to an MPI rank construct, providing the ID number for each node in MPI_COMM_WORLD. When a node performs an MPI send to a remote node it writes to the socket for the remote node, and when a node performs an MPI receive from a remote node it reads from the socket for the remote node.

Slide 21 – 23 Network Latency in Cluster Computing

It was seen (slide 34 in presentation 7) that Amdahl's law including communication overhead is given by

$$S = \frac{1}{(1 - F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)}$$

where

$$F_P = \frac{\text{Parallel instructions}}{\text{All instructions}}$$

$$F_{overhead} = \frac{CPI^{comm}}{CPI} = \frac{\text{Communication activity CC per instruction}}{\text{Computation activity CC per instruction}}$$

In order for cluster computing to work well, F_P and N must be large and $F_{overhead}$ must be small.

In a typical switch-based cluster computer built from standard PC workstations, the total message latency is about 150 μsec (70 μsec for processing in the communications protocol stack in each node and 10 μsec in the switching hardware). On a PC with a 1 GHz clock rate, the clock cycle is 1 nsec = 10^{-3} μsec. The total message latency is then

$$\text{Message latency} = \frac{150 \times 10^{-6} \text{ seconds per message}}{10^{-9} \text{ seconds per clock cycle}} = 1.5 \times 10^{-5} \text{ clock cycles per message}$$

about the time required to handle a page fault in the OS.

The overhead depends on the number of messages passed as the program runs. That is,

CPI^{comm} = clock cycles per message × messages per instruction

where clock cycles per message is the message latency of 1.5×10^5 clock cycles per message. With these parameters, a reasonable level of overhead (F_P < 10%) requires that each node run about 1.5 million instructions between messages.

For this reason, improving the latency of the interconnection network is a critical aspect of cluster computing. The Alpha 21364 (slides 15 – 17) was designed with specialized communication hardware, as have mainframe computers since the 1960s. Specialized hardware for offloading TCP and DMA for message passing is shown on slide 22.

Slide 23 shows a typical cluster configuration using conventional PCs. Each node is a shared memory multicore SMP system running tightly coupled code over OpenMP. The nodes are connected using TCP/IP over Gigabit Ethernet. In the resulting cluster of SMPs (CluMP) nodes exchange data for loosely coupled message passing code over MPI.

Slide 24 – 26 Thunder Cluster

Slide 24 shows the Thunder Cluster, a CluMP machine build for the US Department of Energy in 2004. The machine consists of 1024 Linux nodes, where each node is a quad-core Itanium 2 "Madison Tiger4" SMP. The Itanium is Intel's true 64-bit superscalar CPU, based on a RISC ISA with compiler-based explicit parallelism (slide 67 in presentation 3), virtual registers (slide 30 in presentation 3), and instruction predication (slide 19 in presentation 5). The Thunder reached a peak theoretical performance of 23 TFLOPS (1 TFLOP = 10^{12} floating point operations per second). Additional technical details for the node are found on slide 25. The interconnection network is shown on slide 26. The switching system provides a very high transmission user-to-user bandwidth of 900 MB/s in 5 layers.

Slide 27 – 34 Blue Gene/L

Blue Gene/L was designed as a massively parallel supercomputer for the US Department of Energy. With a peak performance of 596 TFLOPS it was the fastest supercomputer in the world in 2007 (only second fastest in 2008). The system contained a total of 64 Knodes (65,536 nodes) and a total of 32 TB of main memory.

The design of a computer this large creates unusual constraints and requirements, especially regarding equipment costs. Another significant factor is the cost of electrical power to operate the machine. Generally, higher performance requires more electrical power (faster hardware requires more computations per second, which means more binary transitions per second at the

circuit level, which means more electrical current per second). It is impractical to provide more than 20 kilowatts of electrical power to a rack of computer hardware. Together, this means that

Performance per rack = performance per Watt × Watts per rack

depends strongly on **performance per Watt**. In order to achieve high performance per Watt, the designers chose to use medium power processors. While high performance superscalar CPUs with dynamic scheduling deliver high performance (high instruction level parallelism) at high power consumption, Blue Gene/L was designed to provide high thread level parallelism at lower power consumption. The CPU is a highly integrated system on a chip (SOC) based on the IBM Power architecture. Technical details are presented on slides 31 – 33.

An upgrade of the Blue Gene/L design — Blue Gene/Q — was delivered to the US Department of Energy in 2012. It contains 98,304 (96 K) 16-core nodes with a total of 1.6 Petabytes ($1.6 \times 1024 \times 1024$ GB) of main memory.

Slide 35 – 38 Example of Cluster Application

A bad application for a cluster computer is multiplication of matrix **A** and vector **b** (shown on slide 35). This problem works well on a shared memory SMP but is dominated by communication overhead on a cluster.

The data decomposition is based on rows — the manager thread loops through the rows of matrix **A**, sending row_i to of the matrix to node i for execution. Each worker thread receives its row and multiplies it by a local copy of vector **b**. Each worker thread returns its result and the manager thread assembles the results as the vector **Ab**.

For an N component vector, the **computations** are N multiplications and N – 1 additions, for a total of 2N – 1 FLOPs (floating point operations).

The communications overhead is the transfer of the matrix, N^2 floats, and the return of the vector components, N floats, for a total of $N^2 + N$ floats. The communications overhead (estimated as floats sent and received — 2 operations — per flop performed) is about $2(N^2 + N)/2N$ or about N for large N. Under these conditions, the speedup goes to zero. This kind of data decomposition is not practical.

A better example is the numerical integration shown on slide 37. In this case, each node initializes its own data and must only communicate its result. Dividing the integration into N steps, the sequential version performs 2N + 1 FLOPs. In MPI, N can be made as large as the system will allow. Each node performs 6 FLOPs and sends 1 float as its result (causing 1 receive in the manager thread). In this case, with F_P about 1 and N large, the speedup is about 3.

Slide 38 shows applications in which clusters deliver good performance. Thread replicating servers (where a new thread is spawned for each client request) is a good application because there is very little communication between threads. Numerical integration and other forms of number crunching also work well. The applications run on US Department of Energy supercomputers are typically numerical solutions for 2nd order partial differential equations, such as weather forecasting and simulation of nuclear interactions. In such solutions, a large set of data is divided into regions. The computations within each region are independent, except when data on the boundaries between regions is exchanged. After several iterations, the set of independent regional solutions merge at their boundaries to form a solution over the entire set.

Slide 39 – 40 Learning Curve in Parallel Programming

Interesting research has been performed on the learning process for parallel programmers. For small-scale projects, typically "embarrassingly parallel" applications on shared memory SMP systems, skills can be acquired relatively quickly by using standard design rules.

Large-scale projects become more difficult, because debugging consistency is difficult to manage across a team of programmers. Coordination has been found to be a mixture of science and folklore.

Large-scale projects generally requires expertise in programming and the heuristics of the mathematical problem. Programmers become specialists in a specific field of high performance computing (HPC).

Slide 40 shows the results of graduate level (MSc) courses in which students were asked to implement a game. Generally, the parallel version with N = 8 CPUs compares well with the parallel version for N = 1 (showing the power of parallel programming). The N = 8 versions compare less well with the sequential versions of the program, showing the difficulty is identifying the best strategies for data and functional decomposition.