Paper 283-25

An Introduction to Parallel Computing

John E. Bentley, First Union National Bank, Charlotte, North Carolina

Abstract

SMP, MPP, clustered SMP, NUMA, data parallelism, shared-nothing and shared-everything architectures. Understanding these terms and concepts is critical to getting the best performance out of your data warehouse or data mart. It's proven technology today to combine dozens or hundreds of CPUs to build a supercomputer to host terabyte-class data warehouses and data marts hundreds of gigabytes in size. However, largely because SAS® Software prior to Version 8 existed in the "sequential" world (with the exception of SPDS®), most SAS users have never had to become familiar with the parallel computing techniques that make these platforms essential in today's business computing environments. Things are changing fast, though. With Version 8, SAS Software is capable of taking advantage of multiple processors, and third-party vendors now have software that integrates SAS into their own parallel products. This paper presents an overview of parallel computing concepts, terminology, and architectures that should provide SAS users with a working familiarity of the subject. The paper is organized into sections on parallel processing in general, then parallel hardware architectures, followed by parallel software architectures. Users working with very large databases, data warehouses, and data marts may find this paper immediately useful.

Disclaimer: The views and opinions expressed here are those of the author and not those of First Union National Bank. First Union National Bank does not endorse, recommend, or promote any of the computing architectures, platforms, or products referenced in this paper.

Introduction

Suddenly, one-terabyte data warehouses are no longer newsworthy--in mid-December 1999, Deutsche Telecom announced plans for a 100-terabyte warehouse. What makes these terabyte-class data warehouses (and data marts) possible? Advances in computer architecture, database technology, and data storage devices all contribute to the *parallel processing* that makes it possible to query these massive databases and achieve reasonable response times.

I suggest that there's a paradigm shift coming. It's proven technology today to combine dozens or hundreds of central processor units (CPUs) together to build a supercomputer to run queries and perform calculations against a multi-terabyte data warehouse or a data mart hundreds of gigabytes in size. Many companies are doing it and, if done successfully, it's transparent to the users. Users often don't know or particularly care that their data warehouse or data mart is really a network of computers. Most of the time, the only discomfort that they might experience stems from the fact that queries must be written in the "flavor" of Structured Query Language (SQL) supported by their relational database software (RDBMS). However, with the RDBMS libname engines, that's changing too.

For SAS, the move to parallel processing started with the Version 6x SQL Pass-Through Facility. Version 7 did away with the need for SQL Pass-Through by supporting RDBMS engines assigned in the libname statement. These engines transparently convert SAS code to SQL, pass the SQL to the parallel RDBMS for execution, and then either receive the results set as a SAS-format data set or present the results as output. Version 7 included the production release of Application Messaging Services (AMS) to enable programs that form an application to communicate directly or indirectly via a message queue. Version 8 includes the Multi-Process (MP) Connect Facility to exploit local host multi-processor capabilities "by allowing parallel processing of self-contained tasks and the coordination of all the results in the original SAS session".

Without MP Connect, SAS executes *sequentially* one step or procedure at a time on a single processor. (SPDS is an exception and will be touched on later.) Even with the parallel RDBMS engines, the results of a SQL query must be returned to a single processor where the remaining SAS code executes sequentially. If the results set is hundreds of thousands or millions of rows, that's clearly a bottleneck.

Using MP Connect, the DATA step and some PROCs can execute in parallel on the platforms that support it. Already, two software vendors have products that make it possible. A year from now, when MP Connect and AMS are proven technology and moved beyond the early adopters, the impact will be such a boost in processing power that a whole New World of opportunity for exploiting data warehouses and data marts will emerge. Up until now, though, largely because SAS exists in the "sequential" world most users have never had to become familiar with parallel computing. Now is the time to learn.

Parallel Processing

The authors of Parallel Systems in the Data Warehouse compare parallel computing to building a house. The house corresponds to the problem to be solved and workers are the CPUs. There are many different tasks involved in building the house, and to get the job done efficiently the workers must work on the separate tasks in the proper order.

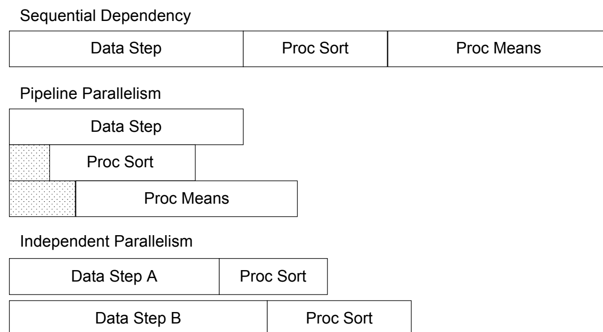
The term *parallel* in the computing context used in this paper refers to simultaneous or concurrent execution—individual tasks being done at the same time. Parallel relational databases such as Informix XPS, IBM DB2 UDB Enterprise-Extended Edition, NCR Teradata, and Sybase IQ12-Multiplex enable parallel query execution via simultaneous and concurrent execution of SQL on separate CPUs, each extracting data from its own disk drive, which holds a portion of the database.

What slows processing is *sequential dependency*, which is a condition in which Task B can't begin until Task A is finished. Native SAS code runs sequentially. Consider a simple SAS program that reads and manipulates a data set, sorts it, and then calculates group summary statistics. PROC SORT doesn't begin until the DATA step ends, and PROC MEANS won't run until the sort finishes. The total execution time for the job is the sum of the times for the separate steps.

A much more efficient execution sequence is *pipeline parallelism.* This is possible when Task B requires output from Task A but it doesn't need all the output before it can begin. In our example, PROC SORT really doesn't need all the data output by the DATA step before it can start. Sorting can start with only two records and then continue by adding and sorting more records as they become available. PROC MEANS can start its calculations as soon as two records of the same group are available from PROC SORT. Because the data flows in a continual stream from one task into another, program execution time is shortened dramatically by pipeline parallelism.

Independent parallelism occurs when there is no dependency between tasks and they can be completed concurrently. Suppose our example has two data sets that need to be read, manipulated, sorted, and then merged before calculating group summary statistics. If we read and manipulate both data sets at the same time and have the processed records feeding into two separate sort operations executing at the same time, then we've achieved both independent parallelism and pipeline parallelism. Independent parallelism combined with pipeline parallelism is clearly the best for minimizing execution time.

Figure 1. Types of Program Execution



Hardware

Enterprise-class computer systems solve mission-critical problems using very large databases. They need to be highly available and reliable (often 24x7x365) and capable of meeting ever increasing performance demands. The primary applications run on computers found in the enterprise data center fall into one of three categories: OLTP, DSS, or business communications:

- OLTP: On-Line Transaction Processing refers to the day-to-day management of business functions, primarily using relational databases. An example is order entry.
- DSS: Decision Support Systems refer to the extraction, analysis, and presentation of data from historic databases to enable knowledge-based decision-making. OLAP, on-line analytical processing, is a subtype of DSS. An example is examining weekly sales over the past quarter by city within state.
- Business communications includes messaging and communication activities, web servers, and document retrieval systems.

Four hardware architectures are currently available for enterprise-class platforms:

- Symmetrical Multiprocessor (SMP) systems: Also known as *shared memory* systems. Multiple CPUs and associated resources running under a single operating system. Memory and disk resources are shared.
- Massively Parallel Processor (MPP) systems: Also known as *distributed memory* systems. Unique instances of the operating system and application running on many physically separate *nodes*, each composed of a single or multiple processors and associated resources. Often no resources are shared between the nodes and communication between the nodes is done by passing messages between the nodes' operating systems.
- Clustered SMP systems: Separate instances of an operating system running on a separate SMP nodes, with each node running an instance of the application and possibly sharing storage devices and data.
- Nonuniform Memory Access (NUMA) systems: Also known as *distributed shared memory* systems. A hybrid of the clustered SMP and MPP platforms

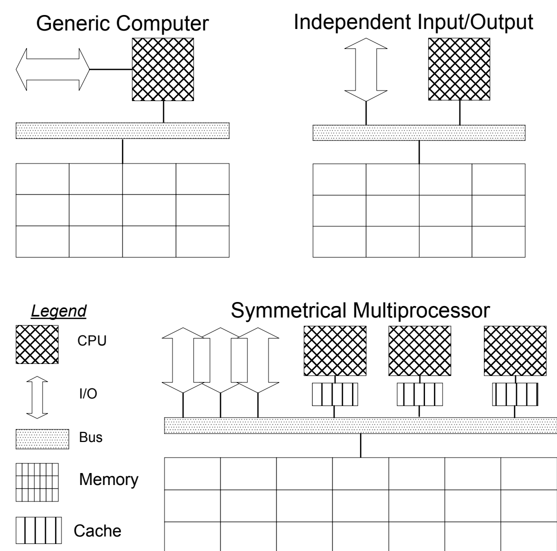
Symmetrical Multiprocessor (SMP) Systems

Probably the most widely used parallel hardware architecture today is the *symmetrical multiprocessor* (SMP), an incredible advance over single CPU systems. All major computer manufacturers make SMP machines, but SAS users may be most familiar with SUN and Hewlett-Packard.

In the original single-CPU systems, the processor was forced to assist in the data input/output operations. While I/O processing was taking place, the CPU couldn't perform calculations and vice versa. Data could only move between the memory and I/O interface via the CPU. The first evolution was to offload I/O to a separate controller. This controller directly interfaces with memory without assistance from the CPU. The CPU only contacts the I/O controller in short bursts to tell it where in memory to place or fetch data from the disk.

When the CPU and I/O controller are simultaneously active, a simple form of independent parallelism is taking place—different, separate operations are occurring concurrently. An important point here is that both components are accessing different portions of the same memory space. This means that a *shared memory architecture* is in place.

Figure 2. Evolution of SMP



Source: Morse and Isaac, Parallel Systems in the Data Warehouse

Once it became possible for more than one device—CPU or I/O controller—to access the same memory, the obvious next step was to allow a number of independent processors and I/O controllers to share a single, larger memory space.

In a *shared memory* architecture, the operating system enforces a strict separation between the concurrent operations. Each of the components—CPU and I/O controller—has a portion of the memory assigned to it and the operating system ensures the logical separation of the assigned spaces so that one component cannot step on the memory allocated to another.

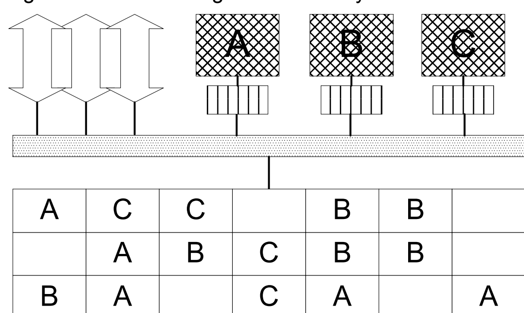
It is important to recognize that in a shared memory architecture the CPUs also share the same *bus* or path to the main memory. This means that only one CPU or I/O controller can be reading or writing to memory at a time. While the bus speed is fast, there is still a finite data transfer rate, called *memory bandwidth*. The operating system is used to manage access to memory, and a technique called *caching*, detailed later, is used to reduce memory contention and improve overall performance.

The most common way that SMP is used is in *multitasking*. While one CPU is executing one program, another CPU executes another. Because the operating system keeps the processes separate and invisible from each other, it appears to each program that it is running on a single-CPU sequential processing machine. This is what allows SAS to run on machines like Sun's multi-CPU Enterprise family of servers.

But what if we have a four CPU system and only one program is executing? Three of the CPUs are, in effect, being wasted. Therefore, while multitasking is useful, it doesn't inherently use all of the power of the SMP architecture. The logical approach is to let all of the CPUs cooperate on a single program (or application). The obvious goal is to make the program execute faster and the assumption is that by putting all four CPUs to work on the same program, the program will execute four times faster. In the same manner, a problem four times as large will complete in the time it takes a single-CPU to solve the original. Applying multiple processors concurrently to the same problem is a basic feature of *parallel processing*.

Figure 3 shows multitasking on a three-processor SMP system, and Figure 4 shows multitasking and parallel processing on the same system. In Figure 3, portions of memory have been assigned to each of the three jobs being run.

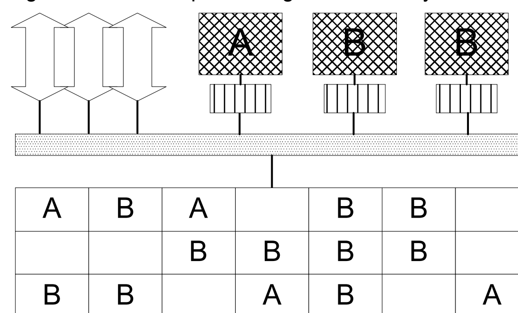
Figure 3. Multitasking on an SMP System



Source: Morse and Isaac, Parallel Systems in the Data Warehouse

In Figure 4, however, only two jobs are running: one CPU is running one job and two are running another. Notice that the single CPU has its own memory space but the cooperating CPUs both have access to the same portions of memory. The logical separation between the two jobs, however, is still being maintained by the operating system.

Figure 4. Parallel processing on an SMP system



Source: Morse and Isaac, Parallel Systems in the Data Warehouse

SMP Strengths and Weaknesses

The commercial success of SMP systems is due largely to their ability to multitask existing (legacy) sequential applications and programs with little or no modification. They do this by not running the application in parallel; in effect, the application thinks it is on a single CPU machine and executes sequentially just like it does on an ordinary single-CPU machine. While one application is executing on one CPU, another application is executing on another CPU. Given the large amount of memory installed on SMP systems—usually 512 megabytes (MB) or more plus a megabyte or more of cache memory—these systems are highly effective at supporting many users running many applications simultaneously, especially OLTP applications.

Parallel processing can be done when the application is written to support it. Some relational databases have been specifically written to use the multiple CPUs available in SMP machines. If the RDBMS supports *multiple threads*, then the CPUs can cooperate as in Figure 4 to speed up query execution. This is accomplished by breaking the code into *parallel* and *sequential regions*. Many processors work on the multiple threads that exist in the parallel regions but only one works when the program is in a sequential region.

The most significant limitation of the SMP architecture is its *scalability*. In short, scalability is the improvement to be had by adding an additional processor to the system. The goals of adding more processors are to (1) execute a job faster (*speed up*) or (2) increase the size of the job that can be completed in a given amount of time (*scale up*). To measure scalability, we first calculate/estimate a ratio of the job size divided by completion time for a single CPU machine. The hope is that the ratio will scale in proportion to the number of processors. We want to see either that doubling the number of processors will double the speed up or the scale up of the machine—ideally it will do both.

Although memory management impacts an SMP machine running in parallel, bus bandwidth limitations are the more important factor that constrains scalability. It is not the amount of memory that does not scale; it's the access to that memory—the bus bandwidth—that does not scale.

SMP Memory Limitations

There is a major disconnect between the speed with which a CPU can process data and the speed with which data can be fed to it. Dynamic random access memory and all its

variants (SDRAM, EDRAM, and CDRAM) simply can't provide data to the CPU fast enough. The solution is to create a *cache* of very fast static random access memory between each CPU and main memory. It's expensive, but better matched to the speed of the CPU.

When the CPU makes a request, the hope is that the data being requested will already reside in the cache, producing a cache hit. If it does, the processor can continue without having to wait for the fetch from main memory. If the data isn't in the cache—a cache miss—then main memory must be accessed and performance suffers. Luckily, caching techniques have been refined to the point where hit rates of 95-99% are common for many database applications.

Caching clearly improves performance. At the same time, though, it creates another problem that degrades performance. If each CPU has its own copy of data in its own cache, what happens when one of the CPUs changes a value in its cached copy, which is then passed through to be written in main memory? The other CPUs must be notified or they will be using dirty data and likely produce an error.

The solution to maintaining *cache coherence* requires that each CPU constantly "listen" to the bus for messages from other CPUs that they are writing to memory. When that happens, the other CPUs have to refresh their cache. As the number of CPUs in the parallel SMP system increases, each CPU must spend more time "bus sniffing" and flushing and refilling its cache.

SMP Bandwidth Limitations

Even in the most cache coherent SMP system, access to the physically shared memory—the bus—is a bottleneck. Once the available bandwidth has been fully utilized, no matter how efficient the operating system is at managing traffic, adding another CPU will not contribute to scalability but will in fact reduce performance. Why? Each additional byte of memory that the new CPU retrieves will come at the expense of a memory reference needed by another CPU. An analogy has been made of hogs at a feeding trough. Once all the space around the trough is taken, no more hogs can be fed. The physical limit has been met.

Although SMP architectures are generally considered not scalable for parallel processing to large configurations—between ten and twenty CPUs based on other system components—SMP vendors like HP and Sun are working to develop technology that raises the scalability threshold for relational databases like Informix and Oracle. At this time, however, most observers agree that SMP systems are more appropriate to hosting 100-gigabyte decision support data marts than they are to terabyte-sized data warehouses.

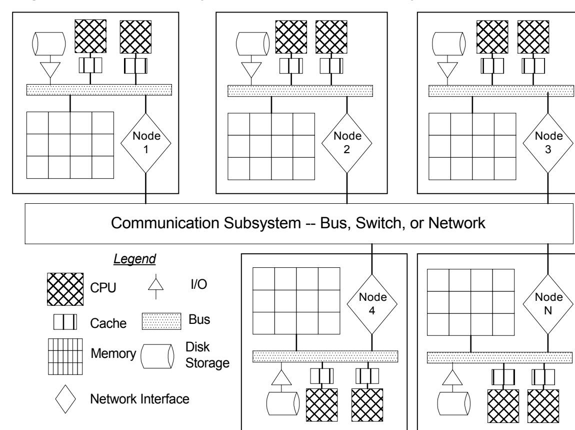
Massively Parallel Processor (MPP) Systems

Massively parallel processor systems solve the problem of SMP scalability. MPP systems have *a distributed memory architecture*, which is the key distinction from the SMP shared memory architecture. Returning to the hogs at the trough analogy, the ideal solution to crowding is to give each hog its own trough—expensive yes, but highly effective for feeding each hog.

An essential concept behind MPP is that each CPU has its own private, nonshared memory instead of contending for bandwidth to the shared memory. This means that as the number of processors increases, both the amount of memory and the memory bandwidth grows. The size of the machine is no longer constrained by the memory bandwidth bottleneck.

An MPP system is composed of *nodes*, which are essentially stand-alone computers each with a CPU, local memory, and I/O and disk capability. Connecting the nodes is a high-*speed communication subsystem* that allows them to send and receive messages and data from each other. Each node has a *network interface* component tying it to the communication subsystem.

Figure 5. Massively Parallel Processor System



A few points must be made about Figure 5. First, the diagram shows only five nodes but MPP systems can have as few as four nodes or several hundred. Because the bandwidth scalability is effectively unlimited, the number of nodes is unlimited. The National Weather Service, for example, is now converting to an IBM RS/6000 SP system with four hundred sixteen nodes to run the complex mathematical models needed for weather and climate forecasting.

Second, the diagram shows a single CPU in each node but multiple CPUs in MPP nodes are not only possible but are in fact increasingly common. The National Weather Service's system has 768 processors. In a later section, we'll discuss the *clustered SMP* architecture.

Finally, although each node is shown as having its own I/O capability, this is not necessarily the case. Most MPP systems have *control nodes* with a higher level of input/output capability. These higher level nodes are also called communication or server nodes, and they have access to all the data stored on all the nodes or a subset of the nodes. Sequential software, like SAS, <u>must</u> be installed on a communication node so that they have access to all the data. Another configuration is that some nodes might not have their own disk storage capability and when they need data, they go through a control node to obtain it from another node. Alternatively, the nodes may have public disks, not private ones. Both of these configurations are forms of *shared-disk architecture* and are discussed later.

The importance of the Programmer

As the term "distributed memory" implies, in an MPP system the memory is distributed usually equally across all the nodes. An important feature is that the data are also distributed across the nodes; in an SMP system—a shared memory architecture—all the CPUs share the same memory and disk space. The key point to remember is that in an MPP system Node 1 usually cannot simply reach into Node N's memory or disk space. If Node 1 needs data from Node N, it is the programmer's responsibility for arranging *messaging* (communication) between the two nodes.

So there is a trade-off for the scalable memory and bandwidth of a distributed memory system—the trade-off is the burden it places on the programmer. Among other things, the programmer, not the operating system, is responsible for insuring that the nodes communicate. Because of the complex features that are needed for a RDBMS to work in a parallel environment, Informix, DB2, Oracle, and Sybase all have a special parallel-enabled version.

A major concern of the parallel RDBMS programmer (and the database architect and administrator) is where the data are located. In our diagrams, think of the memory as a database. In an SMP system, there is only one block of memory—one copy of the database—accessible to all the processors. In an MPP system, though, the memory—and the database—is spread among all the nodes. Each node holds a piece of the database. (This is covered in a later section.)

The physical *partitioning* of the database may be such that a single table is spread among the available nodes. When that happens, each node can only access the rows it holds. Instead of being one large logical table as it is on an SMP system, the table is physically a set of small tables spread across separate computers. Consequently, programming becomes significantly more complicated.

The Communication Subsystem

Data move around an MPP system by means of messages that the programmer causes to be passed, and there are many messages being passed. Think of a forty-node system hosting an 800-gigabyte parallel RDBMS. If one SAS user submits a query for data mining that requires joining sixteen tables, sorting the results set, and then creating a SAS-format data set of 30 million records each with 82 variables, a lot of data is being moved. The potential bottleneck, though, is no longer the memory bandwidth. The potential bottleneck becomes the *communication bandwidth*—the rate at which data can be transferred between the nodes.

As the number of nodes in the system increases, the demand on the communication subsystem also increases. More nodes mean moving more data around the machine, with corresponding stress on the network. In addition, data movement occurs not just because of SQL queries. A monthly data warehouse load process may load 100-gigabytes or more.

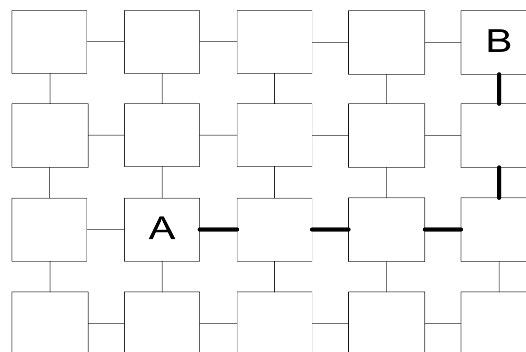
There are a number of ways to set up the communication subsystem. The simplest way is to establish a local area network (LAN) among the nodes, with an Ethernet backbone using TCP/IP for the node-to-node protocol. Practically, however, this approach severely limits the bandwidth and the TCP/IP overhead is high. Using TCP/IP on an Ethernet, a bandwidth of 2 million bytes per second with a latency of tens of milliseconds is common. Fast, but not good enough, so "Gigabit Ethernet" technology is quickly being introduced.

For a high performance MPP, the communication bandwidth should be of hundreds of millions of bytes per second and latency measured in microseconds. *Fibre Channel* is a high-speed networking and peripheral connection technology that moves data at up to one gigabit per second with a 10-microsecond latency. High-end systems, though, usually include a proprietary high-speed. NCR's Bynet-V2 MPP Interconnect provides 120MB per second bandwidth, and Cray systems GigaRing interface delivers a bandwidth of 900MB/sec.

Regardless of the base technology, communications subsystems must be custom engineered and optimized for each installation. Whenever possible, message initialization is implemented in the hardware, as is the switching and routing along the message path. As much work goes in to performance tuning the network as goes into tuning the nodes and the database. When comparing MPP systems, the performance of the communication subsystem is a critical feature and is used by vendors to differentiate their system from their competitors.

The two most common approaches to high-performance communication subsystem are a *point-to-point network* and a *multistage switch network*. The essential idea behind a point-to-point network (PTPN) is that each node has a direct link to a small number of other nodes, its *nearest neighbors*. In a two-dimensional "mesh", each node has four nearest neighbors (north, south, east, and west); in a three-dimensional mesh, each node has six neighbors (N, S, E, W, plus front and back.) In a hypercube, the number of neighbors is $\log_2$*number of nodes.

Figure 6. Two-Dimensional Mesh PTPN



Source: Morse and Isaac, Parallel Systems in the Data Warehouse

If a message from one node in a PTPN needs to go to other than a nearest neighbor then a "bucket brigade" is formed, with intervening nodes on the path passing the message along until it arrives at its destination. From the point of scalability, a PTPN scales along with the number of nodes and, from a pricing standpoint, the cost is constant per node. The disadvantage, though, is that message latency increases with the number of nodes (and the number of messages). This is because not only does the distance—the number of hops—increase as the number of nodes increases, but each node must hold the message for just a fraction of time while it processes it.

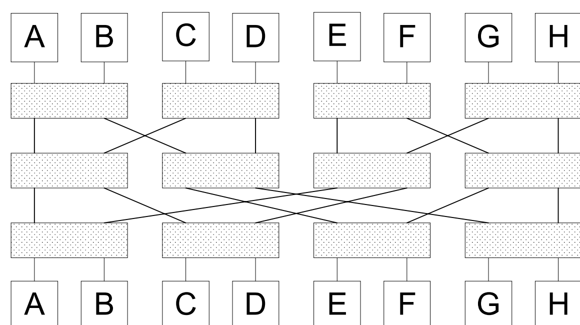
The second problem with PTPN is *message contention*. Two messages with different—or the same—destinations may simultaneously need the same line. Remember that hundreds or thousands of messages may be active at once. When contention occurs, one message must wait and average performance will therefore suffer. This is a place where programming is important. Contention can be reduced if most messages are exchanged between nearest neighbors. This, however, requires that the programmer be aware of the physical layout of the data, which is difficult, may delay program design and development, and reduces the program's portability. A program optimized for a two dimensional mesh may not be appropriate for a different typology. Moreover, if another node is added, the program may have to be modified.

In a multistage-switched network (MSSN), however, there are no concepts of nearest neighbors or bucket brigades. A

message is passed via a series of intermediate *switches* that are physically separate from the processor nodes. The message's source and destination determine the sequence of switches over which it travels, but the number of switches does not necessarily depend on the source and destination because there is no sense of some nodes and switches being closer together than others. The latency for any pair of communicating nodes is the same for all pairs.

A switch has two dimensions: the width, or horizontal dimension, and depth, the vertical dimension. In Figure 7 below, the width is four and the depth is three. Any message must go through three switch elements to go from source to destination.

Figure 7: 2x2 Three-Stage Switched Network



Source: Morse and Isaac, Parallel Systems in the Data Warehouse

As expected, there is a cost to be paid for equal latency among pairs of nodes. The width of an MSSN will naturally increase as the number of nodes increases; the depth only sometimes increases. Increasing the width causes no problems: doubling the number of nodes simply makes the switch twice as wide. However, at a certain point an additional layer of switching will have to be added to the depth.

MSSNs have *magic numbers* that when reached make a significant upgrade necessary. The magic number is the breakpoint where the addition of just one more node will force the entire network to add a level to its depth. In Figure 7, the magic number is eight. Adding a ninth node would require an entire additional level of switching. Doubling the number of nodes will more than double the number of switching elements, with consequent increased costs for the switches themselves plus cabinets, power, cooling, cabling, maintenance and administrative overhead. The physical size of an MSSN scales superlinearly, usually N*log(N) where N is the number of nodes.

Despite the increased complexity and cost of an MSSN, there are major advantages to it. First, the average latency for a message grows very slowly. Adding an additional switch level increases the latency only slightly, nowhere nearly as much as the increase of an addition node in a PTPN. Second, because the distance (measured in switch depth) between pairs of nodes in an MSSN remains the same for all pairs of nodes, the programmer need not be so concerned about the physical layout of the data. Optimizing code does not require a detailed knowledge of the database table locations or partitioning scheme.

Figure 7 shows a very simplified MSSN. Typically, each switch element can support up to sixteen nodes. The deeper the switch, the more different paths from the message source to the destination node. This redundancy improves performance by reducing the likelihood of contention for a given switch port and improves fault tolerance by providing a backup path in case of a port or switch failure.

Distributed Memory Management

Caching plays a significant role within each node of a distributed memory system. As with a shared memory system, a high-speed memory cache sits between the CPU and the node's main memory. The issue of cache coherence, however, usually doesn't exist in a distributed an MPP system because each node is working with its own portion of the database in a *shared-nothing* software architecture. (We'll cover shared-nothing architecture later.) When a shared variable is used by a control node, it is the software programmer's responsibility, not the operating system's, to make sure that all the CPUs know about changes to it.

Each node has its own separate, private, nonshared memory space. The memory available to the other nodes is effectively invisible. This is good in that no other node can interfere with "local" data, but it also means that data held remotely is not directly accessible. When a node needs data from another node, the programmer must use the operating system's messaging facility to arrange data transfer.

"Virtual memory" is usually not used on DM machines because the paging overhead required to write to disk can impose a significant performance hit. Instead, parallel programmers constrain their program to execute within the physical memory available to the node. This means that the amount of physical memory is a critical consideration in system design—if intermediate tables produced by SQL must be held in memory, then the amount of available memory becomes a limiting factor to the size and complexity of the queries that can be effectively run. As a result, MPP systems with one gigabyte of memory per node are not common. The Cray Origin2000 series supports up to 4GB of memory per node, and the IBM RS/6000 SP Power3 SMP supports up to 16GB/node!

There are exceptions, of course. MPP systems are usually designed so that each node has access to a small—two gigabyte—private disk not accessible by any other nodes that can be use to hold a swap file. Data contention and synchronization between nodes is not an issue, and the performance degradation from swapping is minimized by using high-speed I/O such as Fibre Channel.

MPP Strengths and Weaknesses

The physical memory limitations that restrict SMP scalability are clearly not present in MPP systems. Since the design of an MPP system requires that each node possess private, non-shared memory, adding more processors means that additional memory must be installed as well. The memory bandwidth bottleneck doesn't exist.

There is still a potential "bottleneck" for MPP systems, though—can the application make effective use of the processing power and bandwidth available to it? Simply put, if the application can't take advantage of the system's power, what's the point of having the system?

In Parallel Systems in the Data Warehouse, Morse and Isaac suggest that a Darwinian evolution has taken place among parallel hardware vendors in which the "ill fitted" have died out as a result of commercial competition for profitability. In the MPP world, profitability is driven by databases and only systems that show significant performance and cost benefits on database applications

survive. For example, although its name is still on supercomputers, the famous Cray Research was bought out in 1996 by Silicon Graphics when it couldn't sustain commercial viability. More recently, in September 1999 IBM completed its acquisition of Sequent Computer.

The real downside to MPP systems is there are few commercial business needs outside of database applications that require the computing power that MPP provides. Consequently, few other software packages have been *parallelized* to run on this class of computers. Although their cost and scalability make MPP systems ideal for data warehouses and large data marts, functionally they are usually not much more than gigantic database servers.

Clustered SMP Systems

Over the past year or so, vendors introduced commercial MPP systems based on *SMP clusters*. In this architecture, each single-CPU node is replaced with an SMP node hosting two or more processors. The address space for the node still only extends to local memory, and the memory is still shared among the CPUs. The problems of cache coherency and the memory bandwidth bottleneck still exist, of course, but continued advances in memory management, I/O bus architecture, and messaging reduce them significantly.

A *cluster,* then, is a hybrid of the shared memory/SMP and distributed memory/MPP architectures. The basic idea is straightforward: replace each node (single CPU) in an MPP system with an SMP "box". The address space remains distinct to the node and is shared by the node's processors. Message passing via the communication subsystem is used when needed to move data between the nodes. Moreover, code executing within a single node can make use of SMP programming techniques or MPP techniques. Because of these features and capabilities, a clustered SMP is an important and increasingly common variant in MPP architecture.

Clusters pose a distinct challenge to the operating system because the OS must contend simultaneously with both a shared memory architecture within each node and a distributed memory architecture among the nodes. Application programming is challenging as well. For parallel execution, clustered SMP requires that the application is running on all nodes simultaneously, that the node's processors coordinate the local processing and communicate before making changes to shared data, and that all nodes communicate as necessary.

The forces of cost, availability, and scalability have made "pseudo-MPP" SMP-based systems popular. SMP boxes can be scaled up with additional processors and memory and then connected in the MPP fashion without necessarily using the exotic and expensive hardware needed for true MPP systems. With a PTPN network and the 100MB/sec communication subsystem that can be achieved with off-the-shelf networking, this "out of the box" scaling can provide very reasonable performance for more users by providing increased bandwidth and processing power.

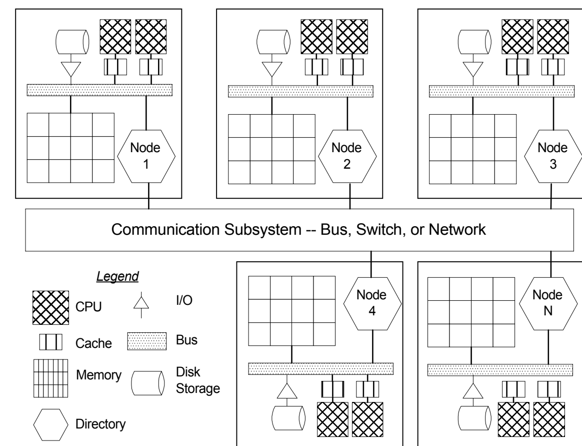
In *shared-disk* database architectures, the performance hit from maintaining cache coherence is magnified in clustered SMP systems, but vendors like HP have learned how to make SMP-based nodes work well together. HP has built SMP systems since the late 1980s and in March 1999 unveiled the HP9000 HyperPlex, which ties together up to sixty-four HP9000 Enterprise Server nodes, each with up to 128 440MHz PA-8500 RISC CPUs, with a proprietary 320MB/sec, 30-microsecond communication subsystem.

Nonuniform Memory Access (NUMA) Systems

We've seen that there is a fundamental trade-off between the ease of programming for SMP and the scalability possible with MPP. SMP has a wide set of commercial application software available but is not able to scale to larger configurations. Commercial MPP applications are limited to databases but the systems have almost unlimited scalability. In an effort to move beyond clustered SMP systems and truly reconcile SMP and MPP, a few vendors have adopted a hardware architecture called *nonuniform memory access* (NUMA) or d*istributed shared memory* (DSM). Implementation differs between the vendors.

A diagram of a NUMA system will easily be recognized as a hybrid between SMP and MPP. Specifically, each node of the systems consists of an SMP machine. As SMP nodes are added, more local memory and bus bandwidth are added to the system as well, so the scalability bottleneck has been broken. NUMA nodes closely resemble clustered SMP nodes with one major difference—each node has a *directory*.

Figure 8. Generic NUMA Architecture



The directory is similar to the MPP node's network interface, but its functionality is much more complex. It is the local manager for a system-wide cache coherence protocol that causes all the nodes to work in concert. Vendors have implemented this protocol differently, but basically each directory maintains a listing of which of its data have been shared with which other nodes. When a "borrower" node modifies a data point "owned" by another node, it notifies the owner node and the owner's directory notifies the other nodes that the data has been changed.

NUMA Memory Access

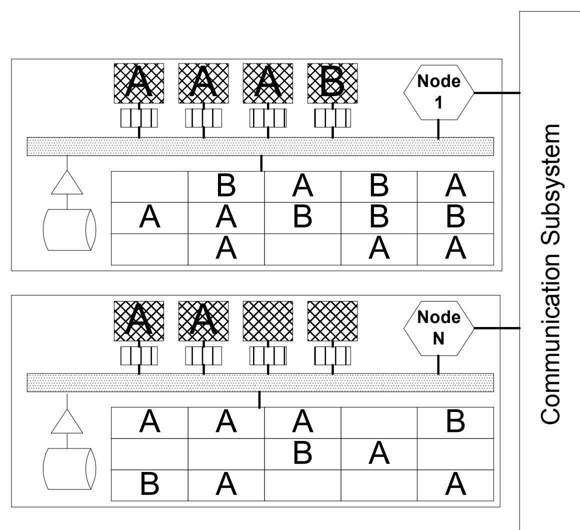
The issue of *memory reference latency* that challenges all computer systems also affects distributed shared memory systems, but with an added twist. When the data already resides in the cache, the fetch into the CPU completes quickly, within one or two machine cycles. Latency-hiding techniques such as pipelining and out-of-order execution minimize total latency so that the average latency is small. But what about when the data isn't in the cache? Then a reference is made to the local DRAM memory, and this requires a bus transaction. Typically, this process takes three to five times longer than if the data were in cache. Again, new technology is reducing this latency, but not to cache reference rates.

In SMP and, depending on the implementation, some MPP systems, the needed data may have been swapped to virtual

memory on disk. Then a page fault occurs, a disk I/O runs and latency can be in milliseconds instead of cycles. To avoid this performance hit, applications are written to avoid using virtual memory, and very large amounts of DRAM are installed in each node.

In a DSM environment, though, a higher logical level of the memory hierarchy exists. This level comes into existence when the data needed does not reside in the local memory of the requesting node but instead resides in the local memory of a different node. When this occurs, the local node sends a request via its directory, a network transaction occurs, and the directory at the remote node intervenes before the data is moved to the memory of the host node. All this adds even more to latency.

Figure 9. Parallel Processing on a NUMA System



Comparing Figure 9 to Figure 3, the difference in latency between *uniform memory access* and *nonuniform memory access* becomes clear. In Figure 3's multitasking SMP, it really does not matter which set of memory blocks are assigned to which CPU. It makes no difference where the data resides--any location is a good as another in terms of memory latency. That is clearly not the case for the DSM in Figure 9. The latency for the memory referenced by a CPU of Node 1 is significantly greater if the data reside on Node N because a network exchange via the directories is needed.

Memory latency increases as the number of nodes increases in a DMA/NUMA system. Vendors of these systems, though, have made incredible strides in writing operating system algorithms that effectively move memory/data and reassign processes to different nodes and CPUs to minimize CPU-to-memory distances and achieve acceptable performance.

NUMA Strengths and Weaknesses

NUMA machines use the SMP operating system as a sort of starting point and then take it to a higher level. Therein lies the system's potential, but, as could be expected, certain SMP problem areas are also taken to a higher level on NUMA. Two SMP issues are magnified by NUMA: *placement, hot spots*. Additional issues that are unique to NUMA include *data migration* and *affinity*.

Because multithreaded parallel relational databases are at the heart of so many data warehouse, data mart, decision support, and data mining applications, *placement* is critical for system performance. A basic task of the operating system is to allocate CPUs and memory to processes—a process must be assigned to a CPU (or multiple CPUs) and memory addresses allocated for use. In SMP, this is simple bookkeeping. Performance will not be improved by choosing among the CPUs or available memory blocks. In NUMA architecture, though, some blocks of memory are closer to one particular CPU than another, so it makes a difference which blocks are assigned where. The OS algorithm must optimize both CPU usage and memory usage by placing processes and data where they will be most efficiently used.

When many logical processes are trying to simultaneously access a single, shared piece of data, we have a *hotspot* at that memory address. A variable collecting an aggregate sum, for example, requires updating by all the CPUs. The operating system controls sequential access to the variable via a lock, and a high level of bus traffic may be generated as all the CPUs constantly check for an unlocked state. In SMP, hotspots are aggravated by the overhead needed to maintain cache coherence but on NUMA systems the need to involve the node directories in accessing shared data not only increases latency but also can dramatically increase bus traffic as the directories talk with one another.

Migration is a NUMA issue that the operating system must address. The node (via the directory) typically holds ownership of shared data on which the data resides. The advantage is that the CPUs located on the owner node will have minimal latencies and generate no network traffic when they access that data. Often though, the owner node is not the node that most uses the data. When that happens, latency worsens and bus traffic grows. To minimize this particular problem, the operating system tracks run-time statistics on which CPUs are accessing which data and *migrates* the data when the CPU-to-memory mapping is no longer efficient. Even with the overhead of moving the data, the gain in reduced memory access latency and network traffic makes it worthwhile. Another approach is for the OS to monitor the processes and threads and maintain their *affinity* with the data by moving them from one CPU to another that is available closer to the data.

Over the past year, NUMA architecture has moved from "bleeding-edge" to proven technology. IBM (through it's acquisition of Sequent) and EMC (through its acquisition of Data General) are two leaders in NUMA systems. A recent article in Information Week (November 29,1999) reported that NUMA technology "will become a linchpin in IBM's data warehousing strategy." In the second-generation NUMA-Q 2000 systems, sixty-four 450MHz Pentium III Xeon processors can reside in a single node, with 64 gigabytes of memory and a 266MB/sec internal bus. A single system can reportedly scale to 256 processors without modifying the application.

Parallelism and Relational Databases

Three principal approaches have been taken by commercial relational database vendors to enable parallel execution: *shared-everything, shared-disk*, and *shared-nothing*. Any of these approaches can be implemented on any of the hardware architectures presented in the earlier section. The match between hardware and software, though, will be better in some cases than others because of performance trade-offs that are fairly clear-cut.

A defining feature of a parallel database is that the tables are distributed across the available disks. Allocating the data in a predetermined manner across the nodes balances the workload and maximizes the memory buses for sequential reads of large, uninterrupted blocks of data.

Shared-Everything Architecture

From a software point-of-view, in a *shared-everything* system (SE) there are two components that can be shared: memory and disk space. With memory, all CPUs will have unlimited access to the same memory block(s), and a single set of global variables, buffers, and counting arrays are declared for all CPUs. A task "work-to-do" list maintained by the *query optimizer* is shared by the CPUs and the next available CPU is assigned the next task on the list. Disk sharing is implemented by allowing any active process to perform I/O operations on data located on any disk in the system regardless of how the data has been allocated across the disks or how the workload is divided among the processors.

Shared-everything processing is accomplished by streaming large blocks of data from the needed tables into memory and then logically subdividing the block and having each available CPU process its assigned subset of data. Aggregates and results sets are summed and combined at the end on a single processor to obtain the final results.

The SE model is a natural choice for SMP. During I/O, the entire bus bandwidth is dedicated to data transfer. Once in memory, partitioning the data is done efficiently because all CPUs have equal access. The memory and bus bandwidth limitations inherent in SMP architecture, however, limit performance and prevent the SE/SMP-style from succeeding on very large databases in a DSS environment.

On distributed memory MPP platforms, the fact that one node's CPU does not address memory in another node prevents the SE approach from being implemented. On a NUMA platform, SE works well provided the I/O and data partitioning operations are modified to use a distributed memory model to minimize traffic across the communication subsystem and to take advantage of the fact that some of the data are already local in reference to the CPU.

Shared-Disk Architecture

Just as the clustered SMP is a hybrid between SMP and MPP architectures, *shared-disk* (SD) is a hybrid between shared-everything and shared-nothing. Like clustered SMP, the justification for implementing an SD approach is often based on practical considerations of risk reduction and prior investment in hardware and software.

In SD, each CPU has its own private non-shared memory, but has connectivity to all available disk storage—memory is private, but disk space is public. This works well for on-line transaction processing (OLTP) where each CPU is likely to be processing a separate unrelated focused query or update, but when querying a very large database for decision support SD is not an advantage. The overhead required for decomposing the data and transferring it into the private memories of the individual CPUs before processing severely degrades performance.

On an SMP platform, a major constraint is that all CPUs are competing for space on the bus between the disk array and the memory. Considering the data stream needed by complex queries, the result is that SE cannot scale to the performance levels needed for decision support using very large databases.

When SD is implemented on a MPP machine where the data are already partitioned across the nodes' disks, the ability of every CPU to access every disk is unnecessary and largely ignored. Otherwise, the system would take a huge performance hit from moving data between nodes and maintaining cache coherence. By having each CPU access only the data on its local disk, a very high level of sustained bandwidth is achieved and little, if any, data is moved between nodes. For an MPP decision support application, SD resembles the shared-nothing architecture.

Shared-disk can fit very well on NUMA platforms. The node-specific memory allows the data to stream from local disk into the CPU(s) assigned for processing when locality is maintained and the directories are designed specifically to coordinate data movement among the nodes. Like MPP though, on NUMA platforms data locality is emphasized during RDBMS implementation, so decision support databases usually resemble the shared-nothing architecture.

Shared-Nothing Architecture

The *shared-nothing* (SN) architecture is not relevant to SMP systems. In a SN configuration, neither the memory local to a particular CPU nor the local disk storage is directly accessible by the other processors. Data movement between nodes is minimized and controls like global aggregation variables are implemented by message passing via the communication subsystem.

For distributed memory/MPP platforms, SN is a natural fit because the data are already efficiently partitioned across the nodes' non-shared disks, ready for streaming into the CPU's local, non-shared memory. The scalable I/O bandwidth of MPP systems makes it possible (theoretically at least) to build machines to handle databases of unlimited size. Shared-nothing also works on NUMA machines, but it requires enforcing local memory and disk and ignoring the hardware's designed-in shared memory and shared-disk capabilities.

The approach taken to partitioning a table across the available disks/nodes takes on an added importance when implementing SN. A common approach is to recreate an empty identical database structure on each node and subset the database into these smaller tables so that each node holds a subset of the database. In effect, each node holds a self-contained database. If the database design holds a customer table and there are 50 nodes, then there would be 50 instances of the customer table, each with a different set of rows. (More on this in the next section.)

To execute a query against this customer table, each node would simultaneously run the query against its own local database, thereby achieving *data parallelism*—simultaneous processing of the data on the node it resides. As the low-level queries complete, a merge phase executes in which the local results are moved to the query's control node for summation or aggregation into a global results set. The control node may or may not be one of the nodes participating in the query.

In data warehousing, shared-nothing architectures predominate. With a shared-nothing RDBMS in place, a "brute force" approach can be used to easily dig through the mountains of data used by decision support queries.

Data Partitioning Across Nodes

The motivating factor for *partitioning* data across the available nodes is to increase the overall bandwidth by spreading the data across many disks, thereby having many sources providing data for any particular table scan. (For this discussion, we'll pretend that each node has only one big disk, and we'll use "node" and "disk" interchangeably.) The total bandwidth available to a table is the sum of the bandwidths of the disks accessing the table in parallel. If, for example, we have a 1GB table on a single disk and the disk's I/O bandwidth is 10 MB/sec then it will take 100

seconds to read the table. If we spread the table across ten disks, though, each disk will hold only 100 MB and a parallel read can be accomplished in only 10 seconds. The total available I/O bandwidth becomes 100 MB/sec.

How the partitioning is done has performance implications for certain types of queries, especially those requiring joins. With the exception of Sybase's IQ database, all major parallel databases use *horizontal partitioning* to split the master data table by rows so that complete rows are stored together within each partition. Sybase IQ uses *vertical partitioning* to split a master table by columns. Vertical partitioning has the advantage of reducing I/O during a join by returning fewer columns, but it often requires additional joins to get all the needed columns, thereby increasing I/O.

With horizontal partitioning, if we have N nodes (disks) and R rows of a table for input, how do we decide on which N to place R? There are three standard approaches available for deciding which rows to put on which disks: *range partitioning, round robin,* and *hashing*.

In *range partitioning*, a subset of the fields of record R is specified as an ordered partitioning key. This key is not necessarily the primary key of the table, but it often is. Each of the N disks is then associated with a range of the key values. The advantage is that the partitioning key acts as a built-in index. The disadvantage is that the data can be skewed to certain disks. If the range partitioning scheme doesn't consider unequal key distribution, then load imbalance will result. Moreover, even if the scheme is adjusted at the initial load, over the long term the distribution of the key may change and eventually it may be necessary to reload the database to rebalance it.

To address the problems of data skewing and imbalance, *round-robin partitioning* simply assigns the "next" row to the "next" disk regardless of the key value. This insures that the rows are spread as evenly as possible and greatly simplifies loading. The downside is that we no longer know which disk holds a given record based on a key value. Even more significantly, there is no way to *co-locate* rows with the same primary key. If we know that two tables are frequently joined using a shared primary key, then it is important to locate the associated rows of the two tables on the same disk. With round-robin partitioning though, associated rows are spread everywhere and any join will likely require moving data between the nodes.

Hash partitioning attempts to have it both ways. Like round-robining, it avoids load imbalance by using a hash function on the partitioning key to randomize the rows and achieve an even distribution. Hashing also insures that rows with identical key values will be placed on the same disk. This way, if the same key assignment and hash functions are applied to two tables, rows that share the same key will be co-located on the same disk.

Parallel Processing and SAS Software

As the size of data warehouses implemented on parallel systems grows and the need for efficient decision support and data mining becomes more acute, users are naturally looking to parallelism for a solution. With Version 8's MP Connect, SAS has been effectively "parallelized" to take advantage of multi-processors on SMP and Clustered SMP systems.

The Scalable Performance Data Server (SPDS) incorporates SMP capabilities to run selected processes in parallel. Most importantly, SPDS implements pipeline read-ahead when concurrently accessing multiple tables, executes in parallel during WHERE-clause processing, index updating and index creation when working with multiple indexes, and provides a parallel "Quicksort" process. With its current release, SPDS uses pass-through SQL to exploit the capabilities of parallel RDBMS.

Two third-party software vendors now have software available that enables large parts of SAS programs to execute in parallel to supplement their own flagship parallel software products. This allows SAS to interface in parallel with parallel databases such as Informix XPS and IBM DB2-UDB Extended Edition. Ab Initio's SAS Analyzer and Torrent Systems' Orchestrator for SAS both make it possible for the DATA step, PROC SORT, and procedures such as PROC FREQ to run in parallel. Both products enable pipeline and independent parallelism and, on shared-nothing database architectures, data parallelism.

Processing the data in place and returning only the final data set of results to the SAS controller node for aggregation yields incredible performance gains on MPP and NUMA systems by dramatically reducing the amount of data that must be moved to the SAS controller node. The downside, though, is that to implement data parallelism a copy of both SAS and either Analyzer or Orchestrator must be installed on each node where SAS is to run in parallel.

For SAS Software, Version 8's RDBMS engines, Application Messaging Services and Multi-Process Connect Facility demonstrate steady progress towards implementing SAS in a native parallel environment so that terabyte-class data warehouses can be fully exploited. The SAS Institute's global leadership in all areas of data warehousing, data mining, decision support, and information delivery combined with their track record of having the right products at the right time leaves no doubt that SAS is also taking a leading role in bringing parallel processing into the commercial mainstream.

References and Resources

Inmon, W.H., et al. (1999). Data Warehouse Performance. New York: Wiley Computer Publishing.

Stephen Morse and David Isaac (1998). Parallel Systems in the Data Warehouse. Upper Saddle River, NJ: Prentice-Hall, Inc.

Sun Microsystems Database Engineering Group (1998). "Data Warehousing Performance with SMP, Cluster, and MPP Architectures".

The Data Warehouse Information Center (one-stop shopping for links to other DW sites.): http://pwp.starnetinc.com/larryg/

Thuraisingham, Bhavani (1999). Data Mining: Technologies, Techniques, Tools, and Trends. New York: CRC Press.

Many vendors have an incredibly valuable selection of White Papers, research documents, technical specifications, and other documentation at their web site: Compaq/Digital, EMC/Data General, Hewlett-Packard, IBM/Sequent, NCR, Silicon Graphics/Cray Research, Sun, Unisys, White Cross Systems, Informix, Oracle, and Sybase.

Contact Information

John E. Bentley
First Union National Bank
301 S. Tryon Street, M-9, NC-0094
Charlotte, NC 28288
704-383-2686 or John.Bentley2@FirstUnion.Com