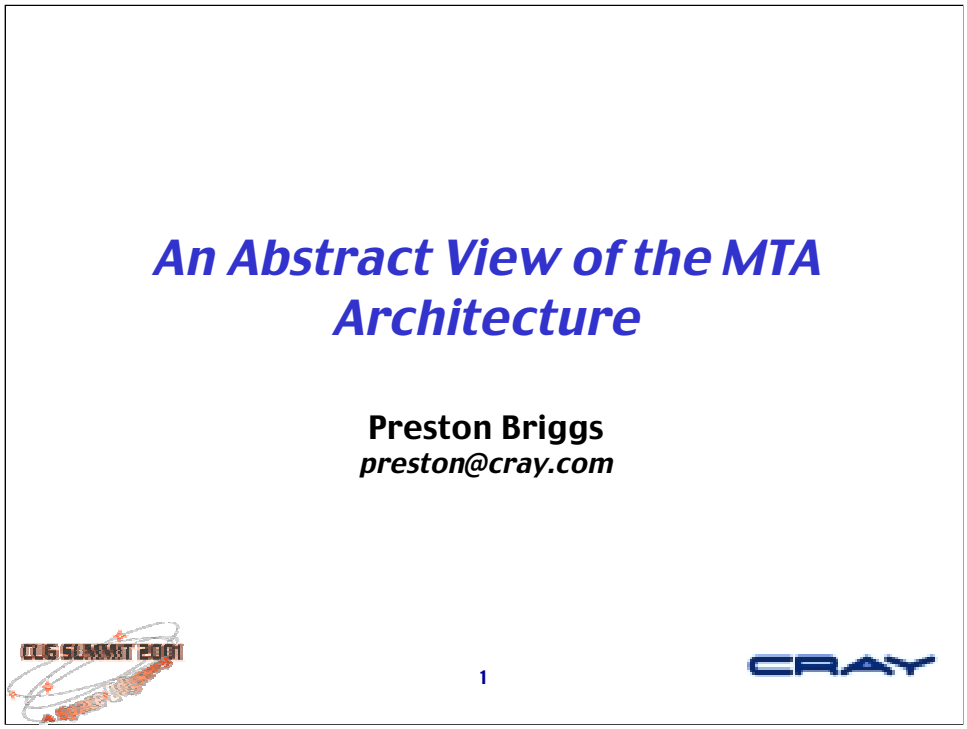


I will begin with a table showing overall system characteristics, to illustrate the scale of machine.

Next is a series of slides, appropriate for those interested in computer architecture, explaining some of the machine's theoretical background.

I follow with a high-level view of the system.

Finally, I will give a more detailed description of each of the major components of the system.

MTA System Characteristics

System size processors	16	32	64	128	256
Peak performance GFlop/s	14.4	28.8	57.6	115.2	230.4
Memory capacity GBytes	64	128	256	512	1024
I/O bandwidth GByte/s	6.4	12.8	25.6	51.2	102.4

CRAY

The system is built out of board, where each board contains a processor, an I/O processor, two memories, and the necessary power supplies and network interface.

As the system scales, more boards are added and the network grows to match.

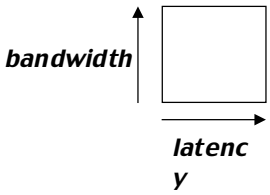
16 boards fit in a cage and 4 cages make a stack.

Architecturally, the current design can scale to 256 processors.

Our future systems will scale to a few thousand processors.

Latency and Bandwidth

In the old days, a "memory unit" could handle one memory reference every tick.



In other words, the latency was 1 tick and the bandwidth was 1 reference/tick.

The next few slides are intended for those interested in the architectural ideas behind the system. Most potential users of the system will be happy to skip over these.

Here, I introduce a graphical language for describing system components

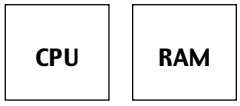
The X axis will show latency expressed in ticks

The Y axis shows bandwidth expressed in reference/tick

This square abstractly represents a memory unit with a latency of 1 tick and a bandwidth of 1 reference/tick.

Balanced Systems

This made it easy to build *balanced* systems



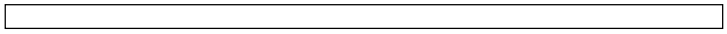
I.e., where the processor and memory can handle memory references at the same speed.

CRAY

The bandwidth of the memory should match the bandwidth of the processor. More memory bandwidth would be wasted and less implies that some programs will wait on memory.

Modern DRAMs

These days, we might expect 1 reference every 16 ticks (or worse).



Here the latency is 16 ticks and the bandwidth is 1/16 reference/tick.

Terrible!

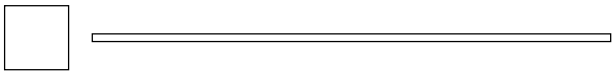
Modern DRAM systems have a much higher latency (in terms of clock ticks) than older systems.

It's not the fault of the memory designers; rather, it's due to the wonderful progress in CPU clock frequencies.

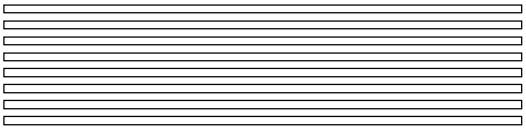
Nevertheless, we must somehow deal with the change.

Improving Bandwidth

One approach is to use a cache (or perhaps several levels of cache).



Another approach is to read many words at time.



CRAY

These approaches work if there is adequate locality.

The cache depends on temporal locality.

The cache illustrated here would have good latency and bandwidth for hits, but bad latency and bandwidth for misses.

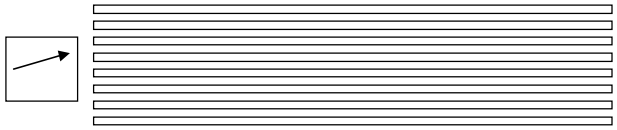
Reading many words at once depends on spatial locality.

In the 2nd picture, we see the same latency, but the bandwidth is improved to 8/16 = 1/2 reference/tick.

It's common to combine these schemes, reading long lines at each cache miss. In this case, the effectiveness of the cache will depend on both forms of locality.

Using Multiple Banks

Our approach is to use multiple banks.



By accepting a small increase in latency, we can approach 1 reference/tick.

Abstracting, we get

That is, a memory with high bandwidth and long latency.

To achieve good bandwidth with DRAMs on the MTA, we use many banks (as do classic Cray vector machines and others).

It works well when we have many banks and references are well distributed among the banks.

The classic vector machines suffer when faced with power-of-2 strides.

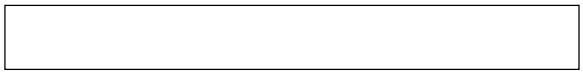
We avoid this difficulty by hashing all addresses, protecting us against any stride problems.

Some Queuing Theory

Little's Law says

concurrency = latency x *bandwidth*

Therefore, in a picture like this



The area of the rectangle represents *concurrency,* or the number of outstanding memory references.

Notice the dimensional analysis works out correctly:

ticks x references/tick = references

High concurrency in the memory indicates that the processor must be able to support a large number of outstanding requests to achieve the potential bandwidth.

In a balanced system, the bandwidth of the memory should equal the bandwidth of the processor, and the latency tolerance of the processor should be greater than the latency of the memory.

Increasing Processor Concurrency

- Allowing multiple outstanding cache misses (especially with long cache lines)
- Vector operations
- Multithreading

CRAY

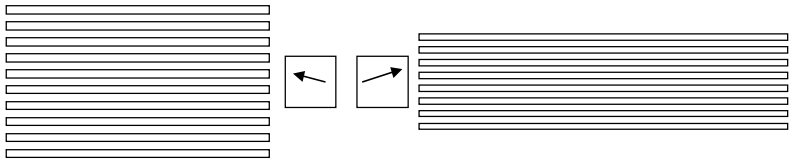
Many current microprocessors allow several outstanding cache misses (perhaps 4 or 8). This seems quite inadequate in the face of the huge latencies to their main memory.

Vector systems are able to issue many loads at once using vector operations, thus tolerating latencies proportional to their vector length. For this reason, vector systems require ever longer vectors to approach peak performance.

Multithreading is a more flexible approach, allowing the tolerance of very large latencies (up to 1024 ticks in the current MTA architecture).

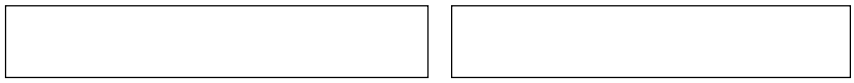
Tolerating Latency

We can tolerate long latencies by using a *multithreaded* processor.



processor *memory*

Abstracting, we get



where the the processor should have more concurrency than the memory.

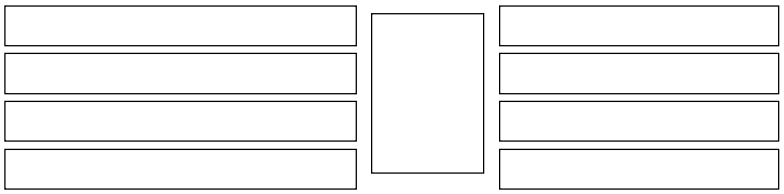
Multithreaded processors achieve latency tolerance by multiplexing between many concurrent threads. If the concurrency of the processor is greater than the concurrency of the memory (the product of its bandwidth and latency), then the processor will be able to tolerate the memory's latency.

Note that the number of threads running in the processor is not equal to the number of banks in the memory.

- The number of memory banks is determined by the latency and the desired bandwidth.
- The number of threads running in the processor is determined by the total concurrency of the memory.

Scaling Up

To achieve high performance, we build a parallel system, with several processors and several memories, all connected with a large network.



processors *network* *memories*

Each time we add a processor capable of issuing a memory reference every tick, we must add a memory capable of handle a reference every tick.

The bandwidth of the network *must* grow in proportion.

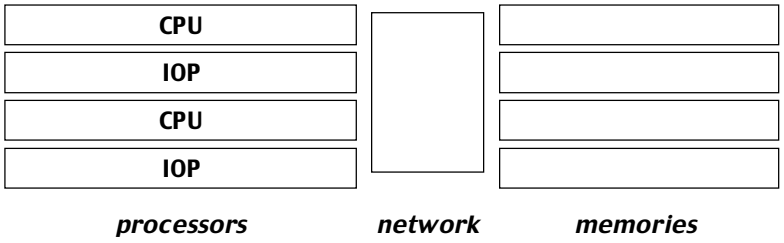
This is our first view of a complete system, with processor, memories, and an interconnection network.

Notice that we maintain our graphical relations, with latencies shown on the X axis and bandwidth shown on the Y axis.

- The bandwidth across the network is equal to the sum of the processors' bandwidths.
- The latency across the network (for our topology) is proportional to $\sqrt{p}$ where p is the number of processors.
- Each processor must be able to tolerate the latency of the memories and the network.

Adding I/O

A complete system needs I/O.



On our system, we add an I/O processor (IOP) for each new CPU.

Since IOPs issue memory references, we must add a memory, too.

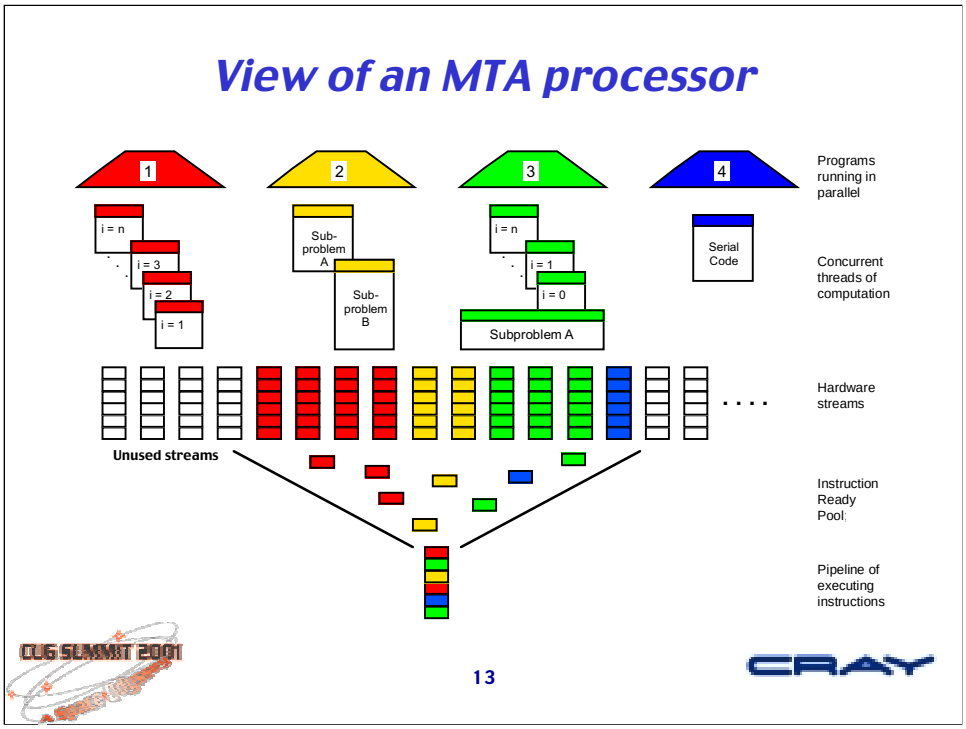
CRAY

The MTA scales in just these proportions. With each new processor, we also add an IOP, 2 memories, and additional network resources.

Each processor is capable of 900 Mflop/s peak and has a bandwidth of 2.4 GByte/s (300 million 8-byte references per second).

Each memory has a capacity of 2 Gbytes and a bandwidth of 2.4 GByte/s

Each IOP has a bidirectional 64-bit HIPPI connection capable of 200 Mbyte/s



This picture illustrates how several programs might be run on a single multithreaded processor.

The top level shows four programs, all running simultaneously on one processor.

The second level shows the various threads comprising each job

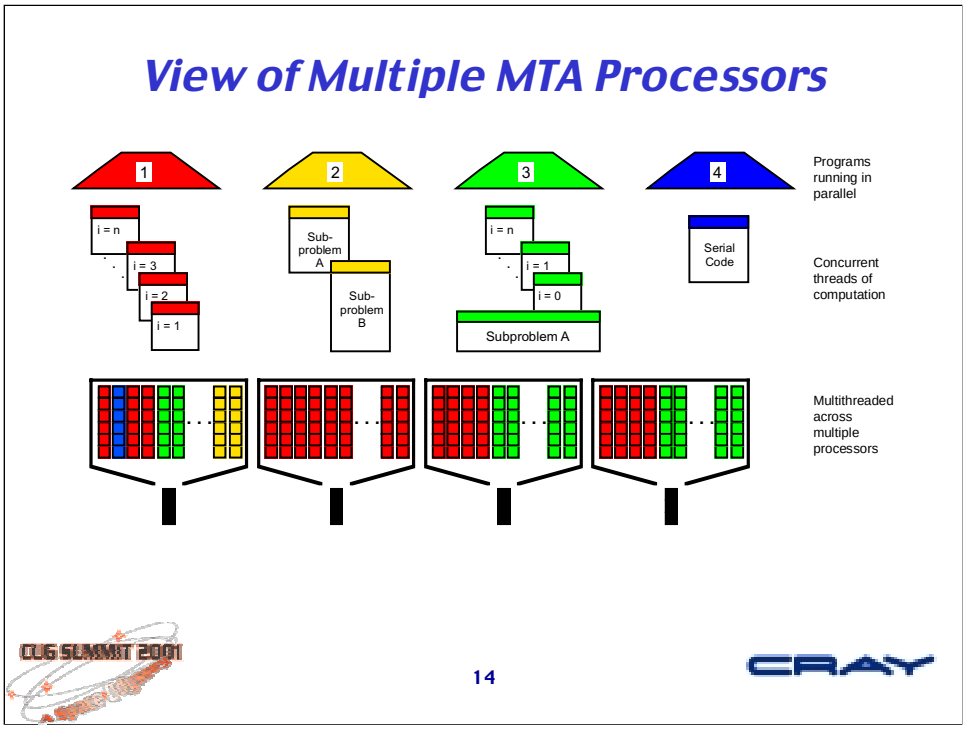
- Program 1 is running a loop in parallel
- Program 2 has two course-grain parallel tasks running
- Program 3 has both course-grain and loop-level parallelism
- Program 4 is serial, although it may exhibit some instruction-level parallelism

The third level illustrates the many streams provided by the hardware. The colored streams are executing instruction on behalf of different active threads. The remaining streams are idle.

The fourth level illustrates the instruction-ready pool. Each active stream can have at most 1 instruction ready at a time.

At the bottom, we see the processor pipeline. There is only one pipeline per processor (21 stages long) and each stream can have only one instruction in the pipeline at a time.

This implies that each stream can issue at most one instruction every 21 clock ticks.



Here we see the same four programs, but spread across a 4-processor machine.

Some programs run entirely on a single processor and some are spread across multiple processors. Of course, the serial program runs on a single stream of a single processor.

Details

Each processor

- Supports 128 streams
- Has 16 protection domains
- Runs at 300 MHz

Each stream

- RISC-like instruction set
- 32 general-purpose registers, 64 bits each
- Three operations per instruction

CRAY

The registers support integer and floating-point operations.
Fastest FP is 64-bit IEEE. We also support 32-bit and 128-bit FP.
Hardware support for soft underflow, etc.

Each instruction has 3 operations

- A memory operation (a load or store)
- An arithmetic operation (including a fused multiply-add)
- A control operation (or an add)

Thus, maximum bandwidth is one memory reference per tick.
Maximum FP is 3 flops per tick = 900 Mflop/s per processor.

For matrix multiplication, asymptotic rate is 600 Mflop/s

Memory Subsystem

- Uniform memory access
  - all memory is global
  - no stride sensitivity
  - logical addresses are hashed to physical addresses
- 2 memories, each 2 Gbyte, per processor
- 1 memory reference per cycle per memory
- Small hot-spot cache in each memory to speed repetitive data accesses

CRAY

Memory is UMA, not NUMA (as far as you can tell)

Every location is equally accessible from each processor

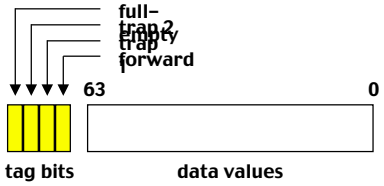
Each memory has 2 Gbytes, with 2 memories per processor

Each memory has 128 banks of memory

The hot-spot cache is a small data cache located on the memory. It's used to help avoid hot spots occurring when many threads attempt to access the same location. Since its located at the memory instead of at the processor, coherency is not an issue.

Memory Details

- 64-bit, byte-addressable words
- Big-endian
- 4 tag bits, plus additional bits for error correction, per word



The extra bits associated with each word of memory are a relatively unique feature.

They are used for several purposes:

- Synchronization
- Data watchpoints for the debugger
- Protecting the malloc's memory pool from erroneous user programs

They may all be accessed and manipulated via ordinary user-level instructions

Synchronized Memory Operations

Each word of memory has an associated *full-empty* bit

- *normal* loads and stores disregard the full-empty bit
- *sync loads* wait for the full-empty bit to be set, then reset it
- *sync stores* wait for the full-empty bit to empty, then set it

sync and *normal* memory operations take the same time.

Waiting consumes no processor cycles.

sync memory operations are available via data declarations and intrinsics in Fortran, C, C++

All communication and synchronization between threads occur in memory.

The MTA supplies cheap and abundant synchronization via full-empty bits associated with each word of memory.

Synchronized loads and stores are no more expensive than ordinary loads and stores.

Lightweight synchronization is a key part of the MTA's ability to exploit parallelism.

The Network

- The network is a 3D mesh
- Packet switched
- Each link runs at 2.4 Gbyte/s
- Resources (processors and memories) are placed at nodes, sparsely, throughout the network.

CRAY

Users are typically unconcerned with the network.

To architects, it's interesting to note that as the machine scales, the network must grow at a faster rate.

Ease of Programming

It's easier to program the MTA for high performance than any other system.

- No message passing
- No data distribution
- No cache management
- No stride problems
- No vectorization

CRAY

As a result of the architecture, we believe the system offers two big advantages: scalability and ease of programming.

Of course, distributed-memory machines scale too, but the programming effort can be very high and many applications can't be made to run well, regardless of the programming effort.

The ease of programming is reflected in the users' efforts, but also in the ability of the compiler to be helpful. Our compilers are powerful precisely because the machine is easy to program.

Larry Carter, a professor at SDSC, says "A day of tuning for the MTA is worth a month of tuning for a distributed-memory system."

One parallel loop is enough to completely saturate the machine. There's no need for multiple levels of parallelism, parallelism plus cache, parallelism plus data distribution, *etc.*