

**CRAY**

POWERED!BYEXPERIENCE



**Productivity  
is a  
11 o'clock tee time**

*John Feo*

*Cray Inc*

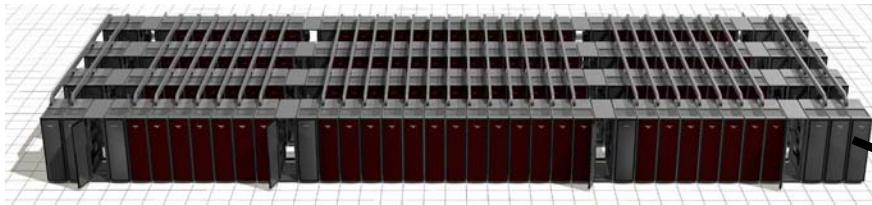
# I want ... I want ... I want ...

- I want to use the best algorithm for the problem
- I don't want my choices restricted by the architecture
- I want the compiler and runtime to implement efficiently a wide variety of parallel techniques
- I want to make use of the hundreds of years of research in parallel algorithms and complexity theory
- ... after I have all this, I'll want a high level programming language, good program analysis tools, and a useable debugger.

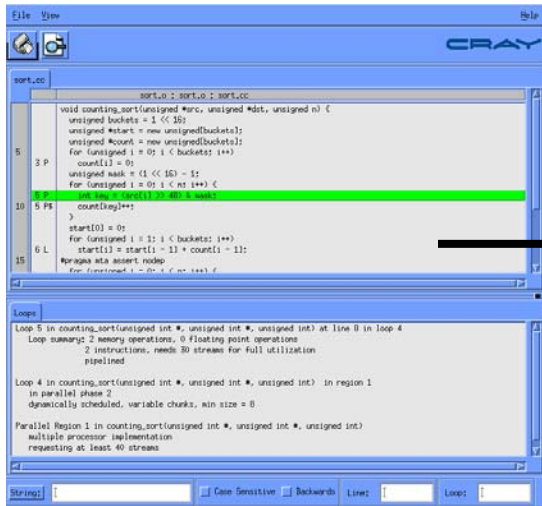
# A minimum set of techniques

- Data parallelism
- Task parallelism
- Recursion
- Dataflow
- PRAM
- Cyclic reduction
- Branch-and-bound
- Dynamic programming

# Eldorado Overview



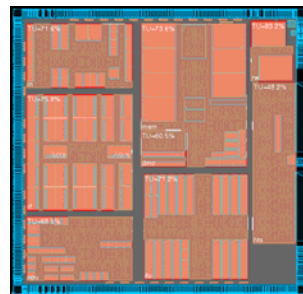
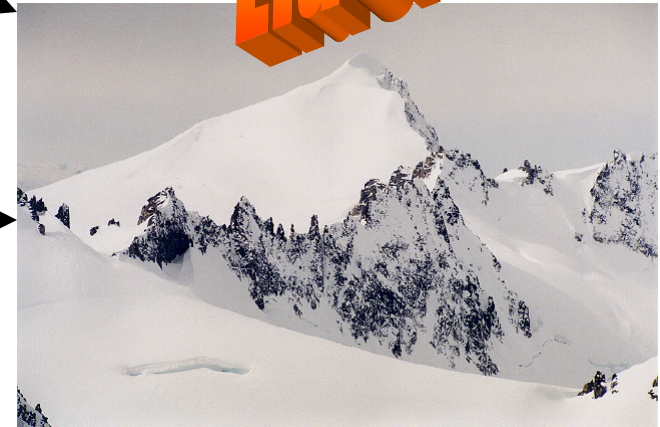
Cray XT3 System Infrastructure



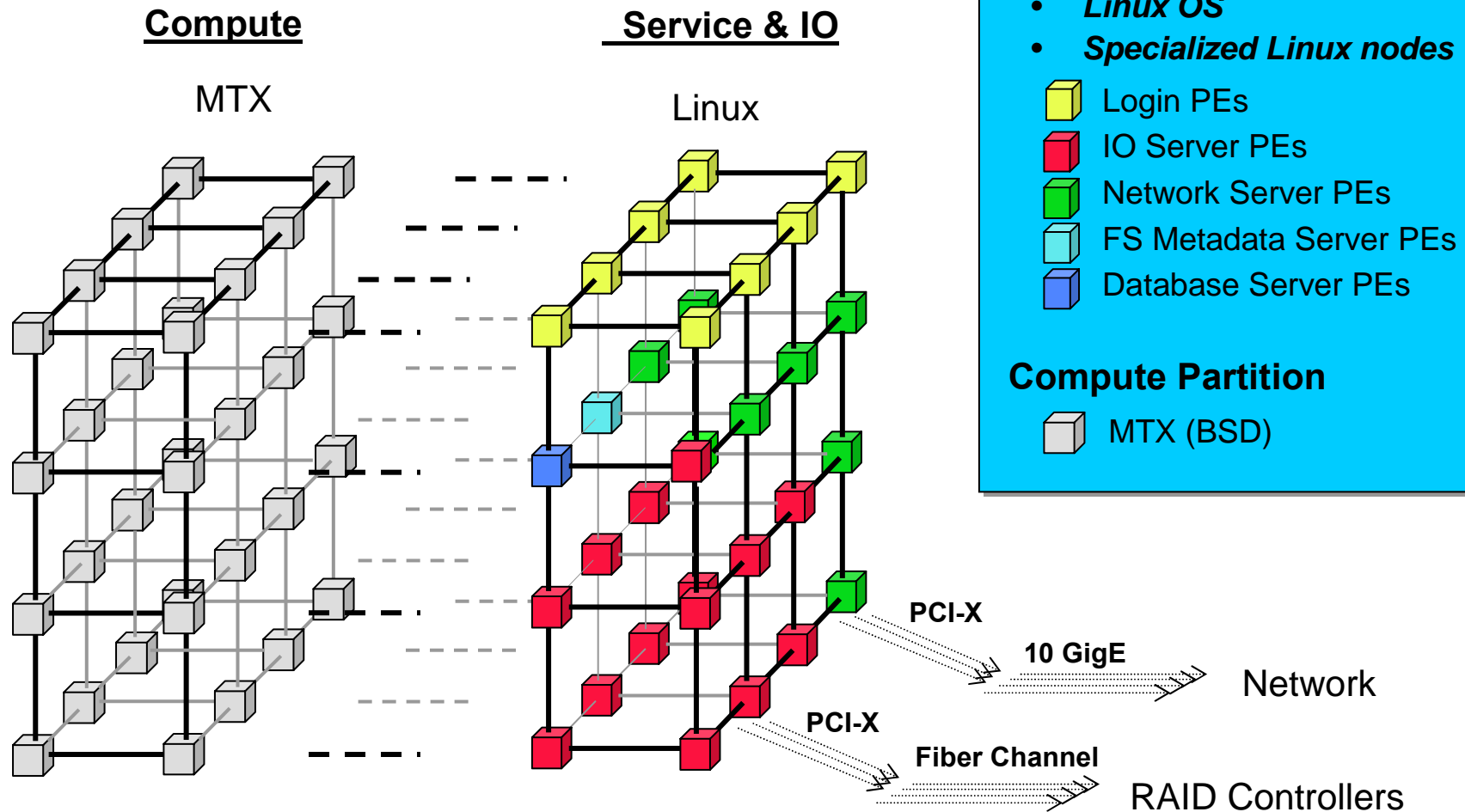
MTA Software

A new multithreaded CPU and minor additions to Seastar

# Eldorado

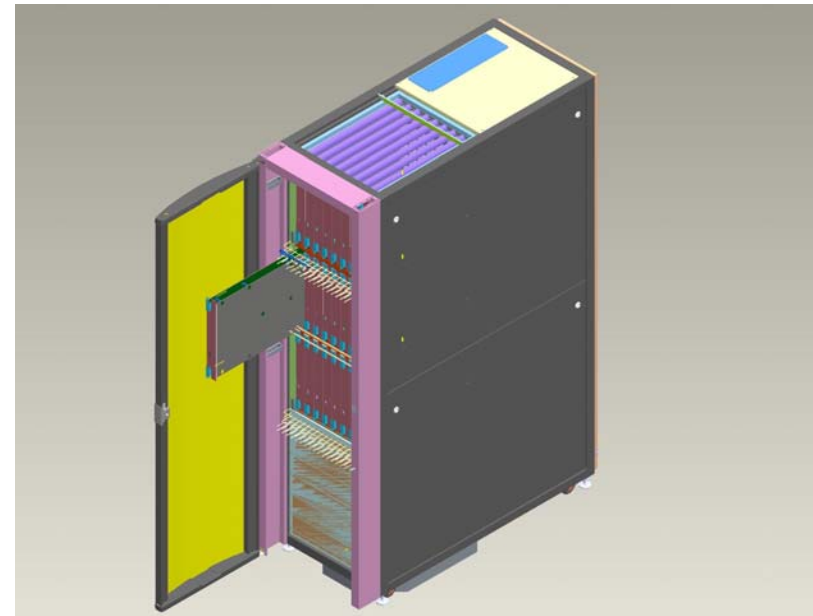


# Eldorado system architecture



# Configurations

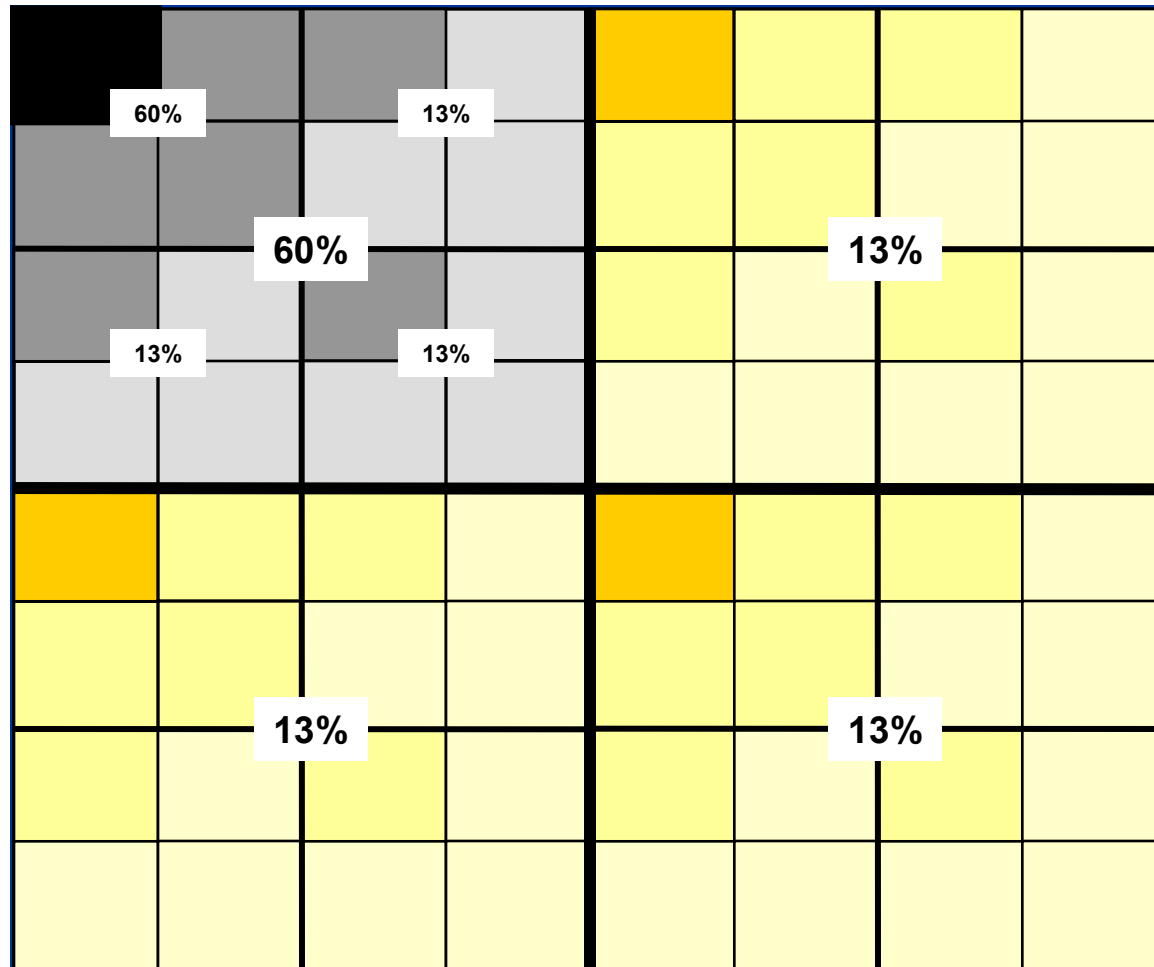
- Cabinet
  - Unit of system growth
- Compute blades
  - 4 Threadstorm processors
  - 4, 8, or 16 GB per processor
  - 4 interfaces to HSN
- Service blades
  - 2 Opteron processors
  - 4 interfaces to HSN; 2 do not inject into HSN
  - 4 PCI-X interfaces
- Blade types can be mixed within a cabinet
- System maximum 128 TB of shared memory



- Graph problem to evaluate HPCS systems
- Power law graph created by the RMAT algorithm
- Kernel 1 creates a graph data structure without self- and duplicate edges
- Kernel 2 identifies maximum weight edges
- Kernel 3 returns subgraphs that include the maximum weight edges
- Kernel 4 identifies the set of vertices in the graph with the highest betweenness centrality score

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

# RMAT



# SSCA 2 graph

- 8M nodes, 64M edges
- Power law graph
  - Node 0 has more than 100K connections (average is 8)
  - Load imbalance is severe
- Need to eliminate self- and duplicate edges
  - Connectivity matrix is impractical in space
  - Sorting edges is practical in space and time if ...
    - ◆ parallel
    - ◆ balanced
    - ◆ at worse  $O(N \log N)$
    - ◆ two step process – sort start vertex, then end vertex

# Sorting start nodes

- Sort  $N$  integer values  $v_i$ ,  $0 \leq v_i \leq R$ , where  $R < N$

**Textbook says Bucket Sort is best**

**Bucket Sort (*src* → *dst*)**

*Count the number of elements in each **bucket***

*Calculate the **start** of each bucket in **dst***

*Copy elements from **src** to **dst**, placing each element in correct segment*

# Step 1

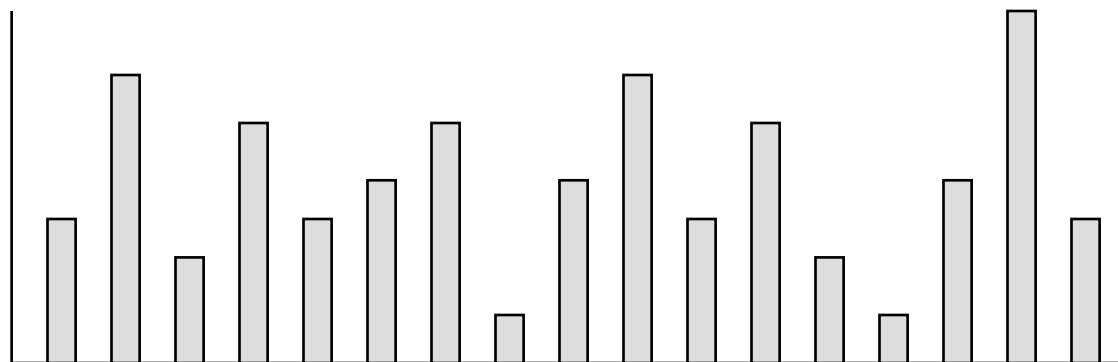


```
for (i = 0; i < R; i++) count[i] = 0;  
for (i = 0; i < N; i++) count[src[i]] ++;
```

*src*



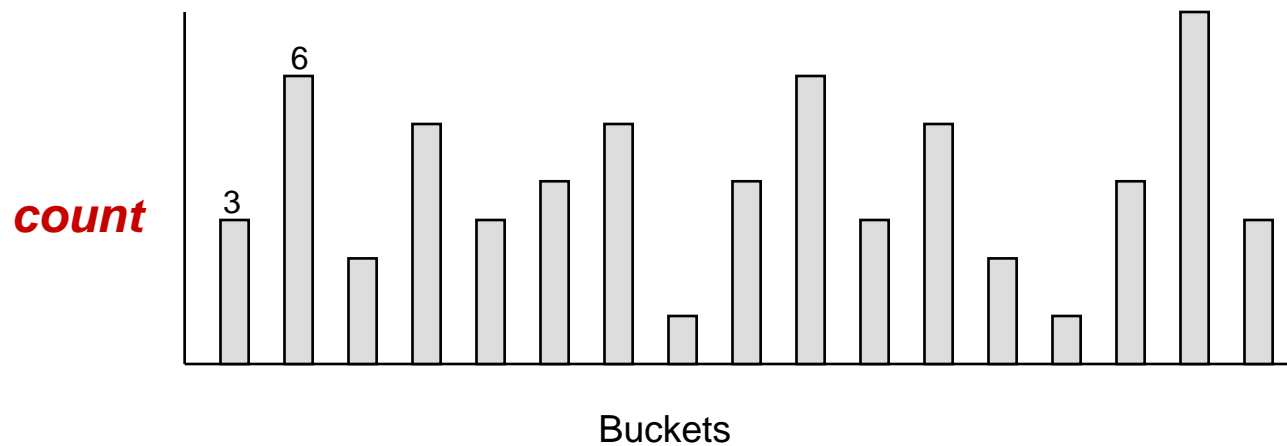
*count*



Buckets

## Step 2

```
start[0] = 0;  
for (i = 1; i < R; i++)  
    start[i] = start[i-1] + count[i-1];
```



*start*

0	3	9	14	15	22	24	26	32
---	---	---	----	----	----	----	----	----

# Step 3

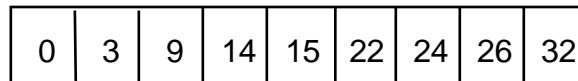


```
#pragma mta assert no dependence
for (i = 0; i < N; i++) {
    index = start[src[i]]++;
    dst[index] = src[i];
}
```

*src*



*start*



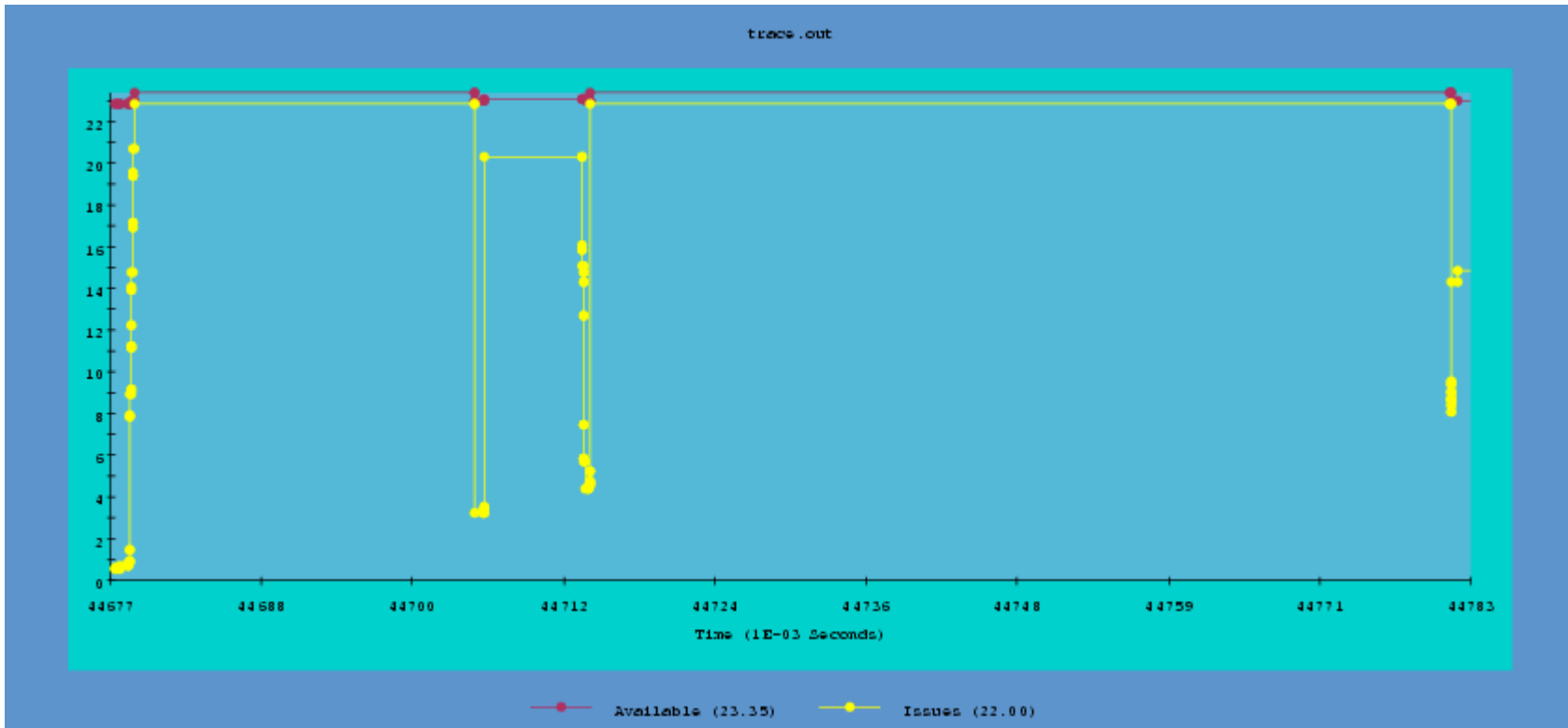
*dst*



# Implementing parallelism

- Responsibility of compiler and runtime system
  - **Programmers should be algorithm experts, not system experts**
- MTA compiler parallelizes automatically the three steps
  - Data parallel
  - Cyclic reduction
  - Atomic reduction and insertion operations
- Runtime system requests, reserves, schedules, and frees hardware resources

# Execution trace for BucketSort

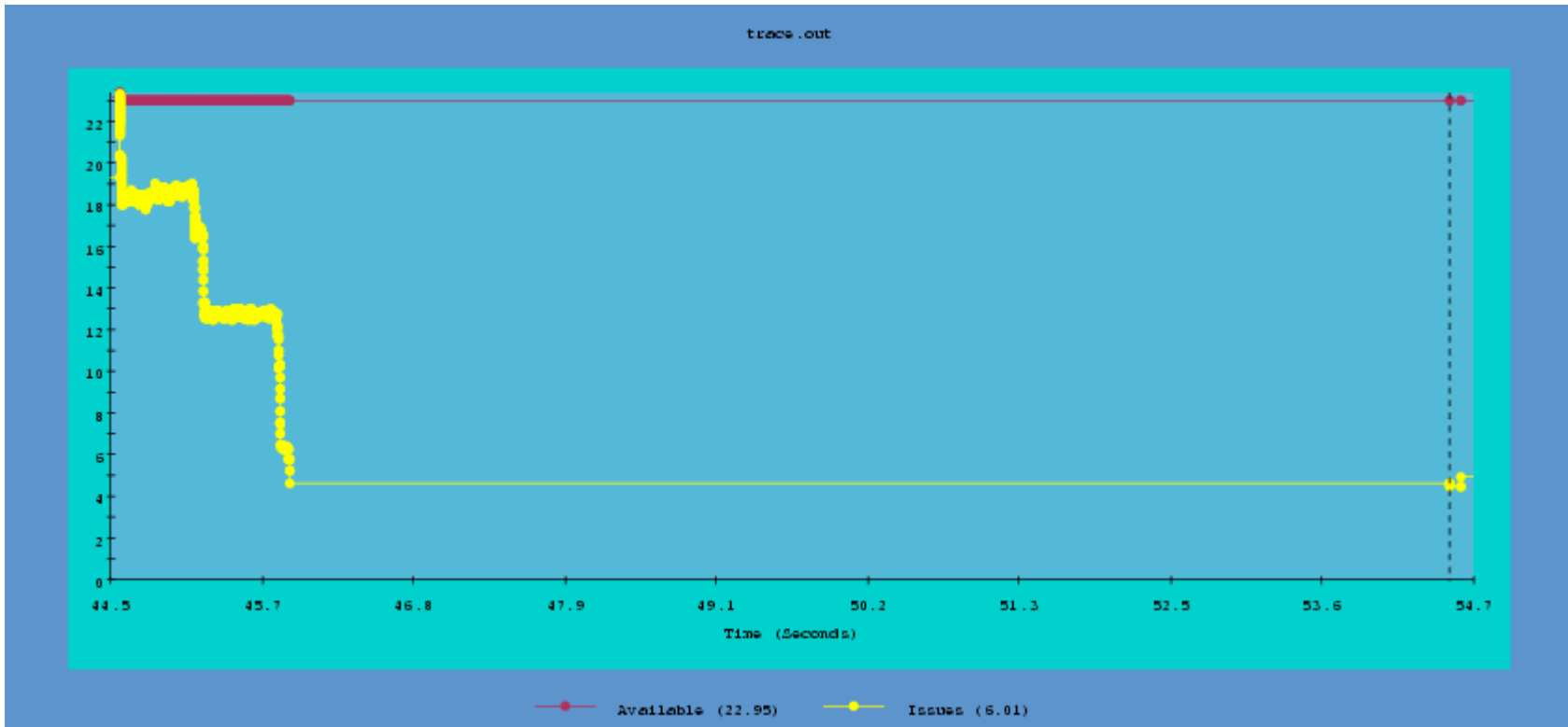


# Sorting end nodes

- For each start node, sort its end nodes
  - Use a dynamic schedule to reduce load imbalance
- Since  $N \ll R$ , BucketSort is not a good choice
- Any in-place,  $O(N \log N)$  is okay
  - Quicksort

```
#pragma mta assert parallel
#pragma mta dynamic schedule
for (i = 0; i < NV; i++) SortEnd(ev, count[i], count[i+1]);
```

# Loop over QuickSort



```
#pragma mta assert parallel
#pragma mta loop futures
for (i = 0; i < NV; i++) SortEnd(ev, count[i], count[i+1]);
```

```
void SortEnd(int *ev, int LL, int UL)
{ int lp, up, pp, pivot;
  future int left;

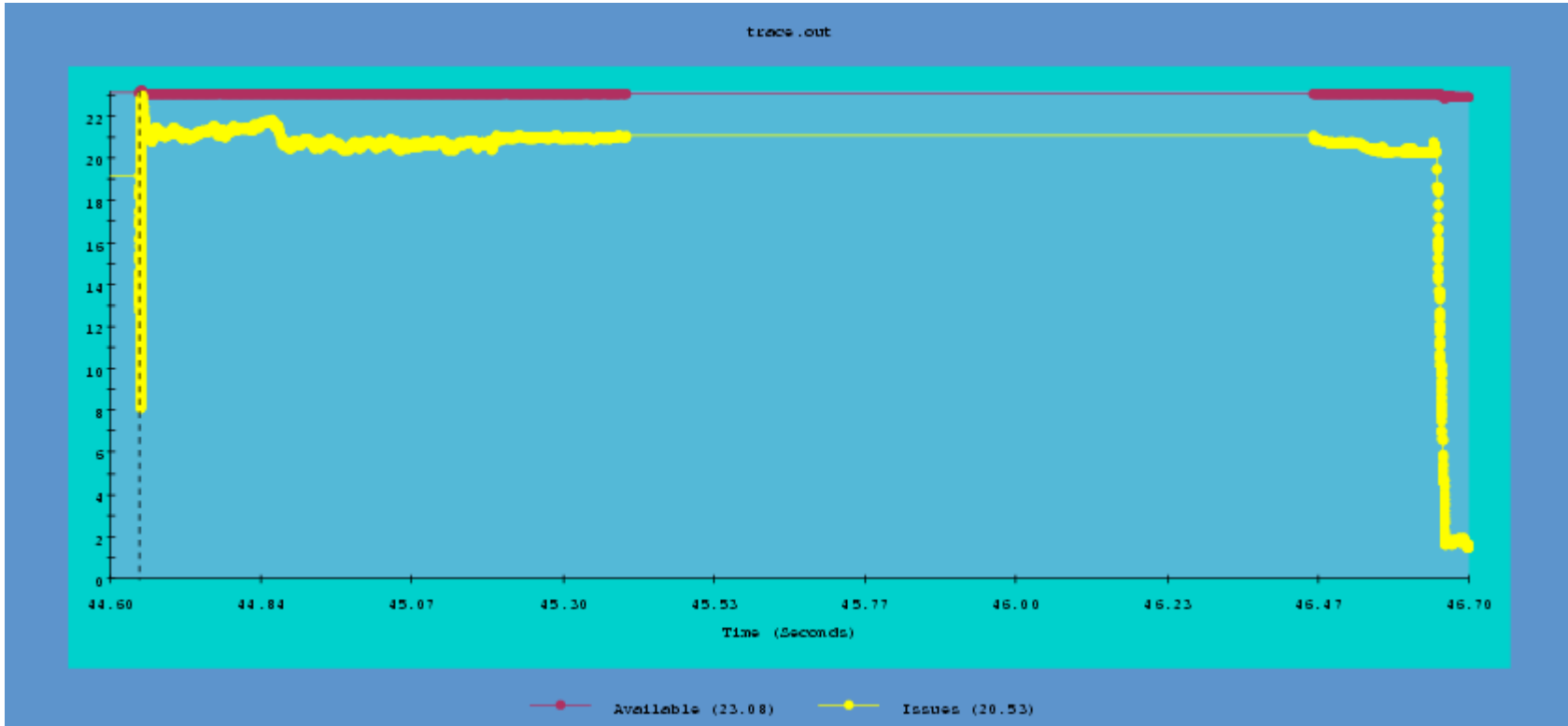
  if (LL >= UL) return;

  if (LL == UL - 1) {
    if (ev[LL] > ev[UL]) swap(ev + LL, ev + UL);
    return;
  }

  . . . pivot elements in place . . .

  future left (ev, LL, pp) {SortEnd(ev, LL, pp - 1); return 1;}
  SortEnd(ev, up + 1, UL);
  touch (&left);
}
```

# QuickSort with futures



# “It’s the hardware, stupid!”



- Productivity is all about hardware
  - Languages and tools are secondary
- A productive parallel system supports a wide variety of parallel methods and programming techniques
- Minimum system requirements
  - Shared memory
  - Multi-threaded processors
  - Use parallelism to tolerate latencies
  - Zero cost single word synchronization
  - Low overhead, dynamic thread creation and resource scheduling