# Fine-Grain Communication

Burton Smith
Technical Fellow
Microsoft Corporation

# Communication Among Cores

- …argues for message passing in shared memory
  - In what might be called “data flow style”
  - Fine granularity helps both parallelism and locality
- …should rely on task scheduling at user level
  - To make blocking and wakeup inexpensive
  - To enable custom protocols and scheduling
- …can benefit from hardware improvements
  - To avoid wasting resources and power
  - To fully leverage hardware multithreading
  - To enable heterogeneous processors

# Rewards Of Anonymity

"On the Internet, nobody knows you're a dog."

…and in shared memory, nobody knows you're a

# User Level Work Scheduling

- A user-level run-time can schedule its cores
  - Tasks that block surrender the hardware thread
  - So do OS calls that may block (nearly all of them)
- The raw OS interface is asynchronous
  - But it appears synchronous to the application
- The OS allocates resources to applications
  - Based on each application's measured ability to use the resources and its targeted performance

    See Bird, S. L. and B. J. Smith, "PACORA: Performance Aware Convex Optimization for Resource Allocation", Proc. HotPar '11

# Messages Via Shared Memory

- Caches may be hardware coherent or not
- Memory access may be uniform or not
- Producer-Consumer synchronization is basic
  - Many other abstractions can be built using it
- One-item buffers (M-structures) are easy to do
  - This amounts to “full-empty bit” synchronization
  - We won’t need an extra bit per memory word
- Two-phase waiting is recommended
  - Spin for a while to amortize overhead, then block

# Example: Prefix Computation

- Given an associative binary operation ($\cdot$) and a sequence $x_0, x_1, x_2, \ldots$ of values from its domain, the $\cdot$ prefixes of the sequence are:

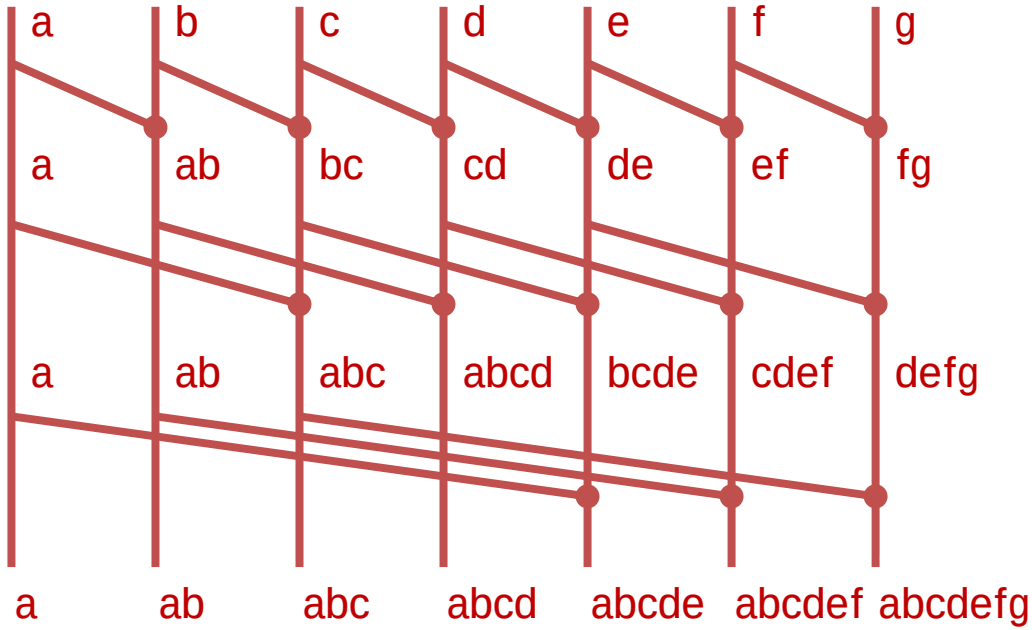
  $x_0, x_0 \cdot x_1, x_0 \cdot x_1 \cdot x_2, \ldots$

  - A variant starts with $\cdot$'s identity element (if it has one)
  - A prefix computation is also known as a "scan"
- Example:

```
a[0] = b[0];
for (i = 1; i < n; i++){
   a[i]=a[i-1]+b[i];
}
```

- A *parallel prefix computation* does this in parallel

# Cyclic Reduction



# Serial Cyclic Reduction

```
int n;
double x[n];
for(int s = 1; s < n; s *= 2){
  for(int j = 0; j < n; j++){
    if(j >= s){x[j] += x[j-s];}
  }
}
```

# Parallel Cyclic Reduction

```
int n;
double x[n];
volatile PCvar<double> xp[n][log2n];
int i = 0;
for(int s = 1; s < n; s *= 2, i++){
  forall(int j = 0; j < n; j++){
    if(j < n-s){xp[j][i] = x[j];}
    if(j >= s){x[j] += xp[j-s][i];}
  }
}
```

# Other Producer-Consumer Uses

- Streaming (pipelining) in an expression graph
  - Loop pipelining (DoPipe)
  - Loop skewing (DoAcross)
- General atomic memory operations (AMOs)
  - Floating point add to memory, for example
  - Each task consumes the old and produces the new
- Locks
  - either *full* or *empty* can mean *locked*
  - If *full* means *locked*, the value can name the holder

# Single Assignment Variables

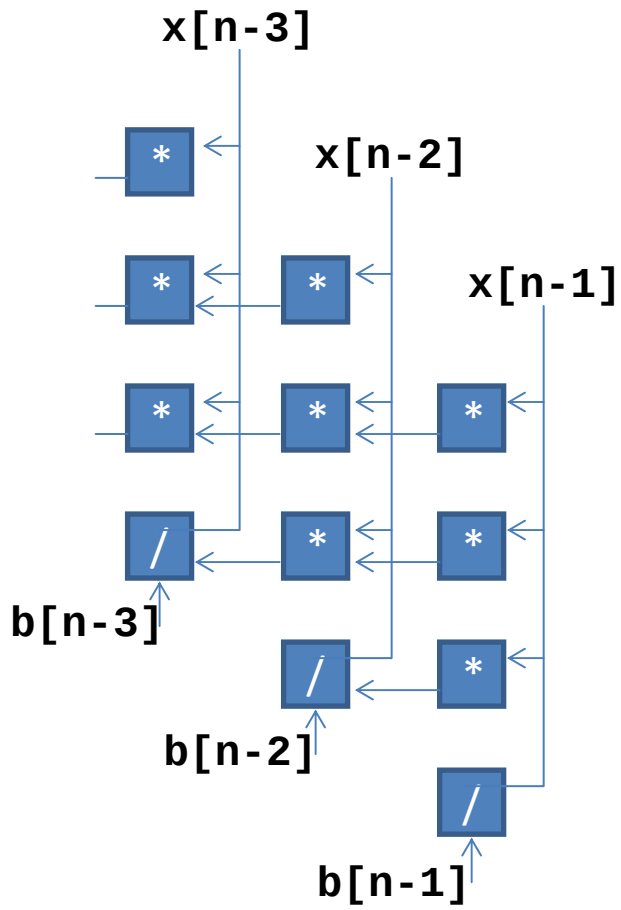
- Known as "I-structures" in data flow-speak
- A single store communicates to all loads
  - Loads that arrive prematurely will wait
  - The store satisfies all waiting loads immediately
  - More than one store is a runtime error
- Broadcasts can exploit this machinery
  - Even when readers dynamically select locations
- Garbage collection is used to reclaim them
  - Liveness analysis *a la* Sisal can reduce GC work

# Example: Back Substitution

- Suppose we want to solve a linear system of equations $Ax = b$
- We factor $A$ in some way, *e.g.* $A = QR$ where $Q$ is orthogonal and $R$ is upper triangular
- Now $Ax = QRx = b$ and we pre-multiply both sides by $Q' = Q^{-1}$ yielding $Rx = Q'b$
- Finally, since $R$ is upper triangular, we solve for $x$ one element at a time starting with the last

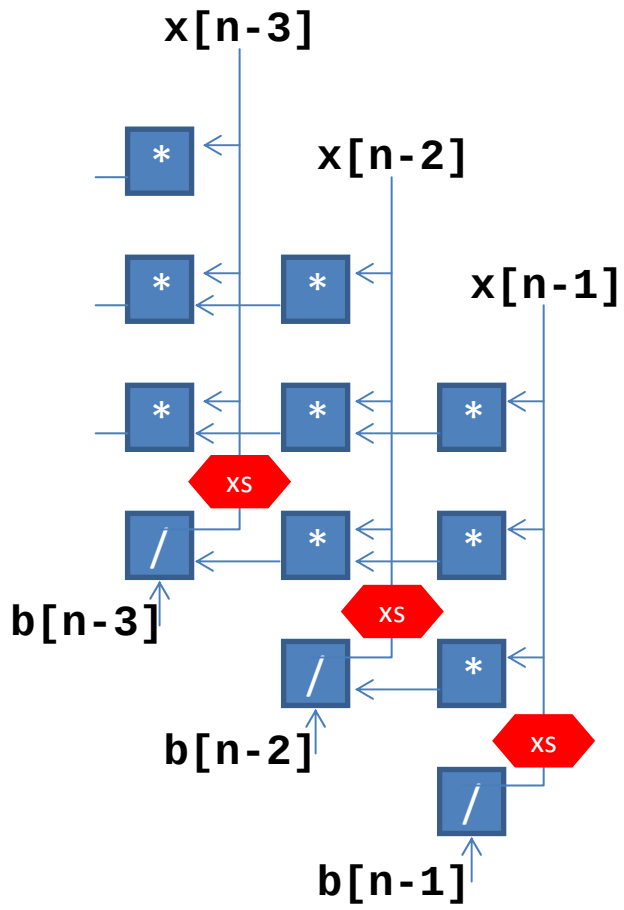
# Serial Back Substitution

```
void backsub(int n,double b[],
                  double R[][],double x[]){
  for(i = n-1; i >= 0; i--){
    double ac = b[i];
    for(j = n-1; j > i; j--){
      ac -= R[i][j]*x[j];
    }
    x[i] = ac/R[i][i];
  }
}
```

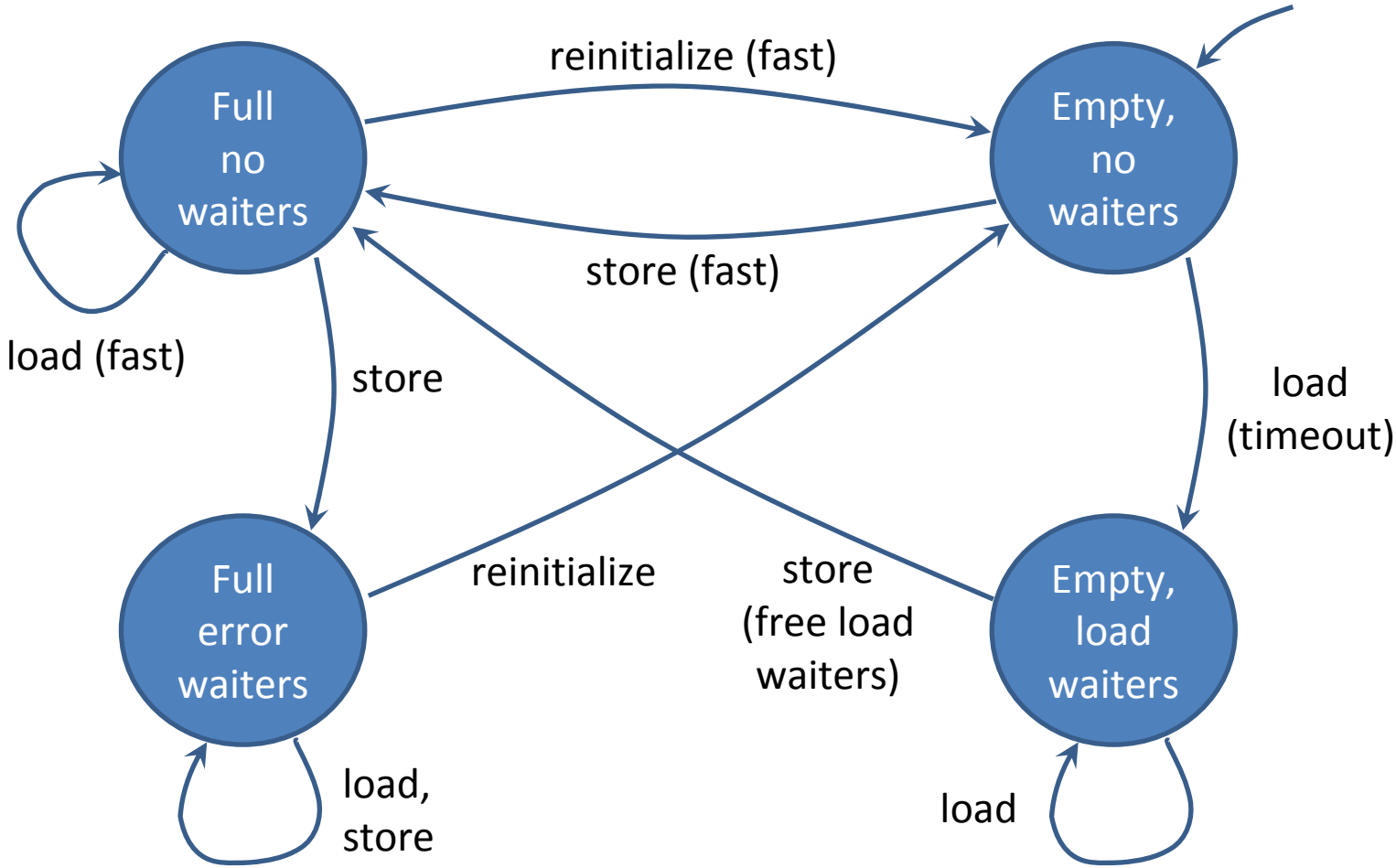


# Parallel Back Substitution

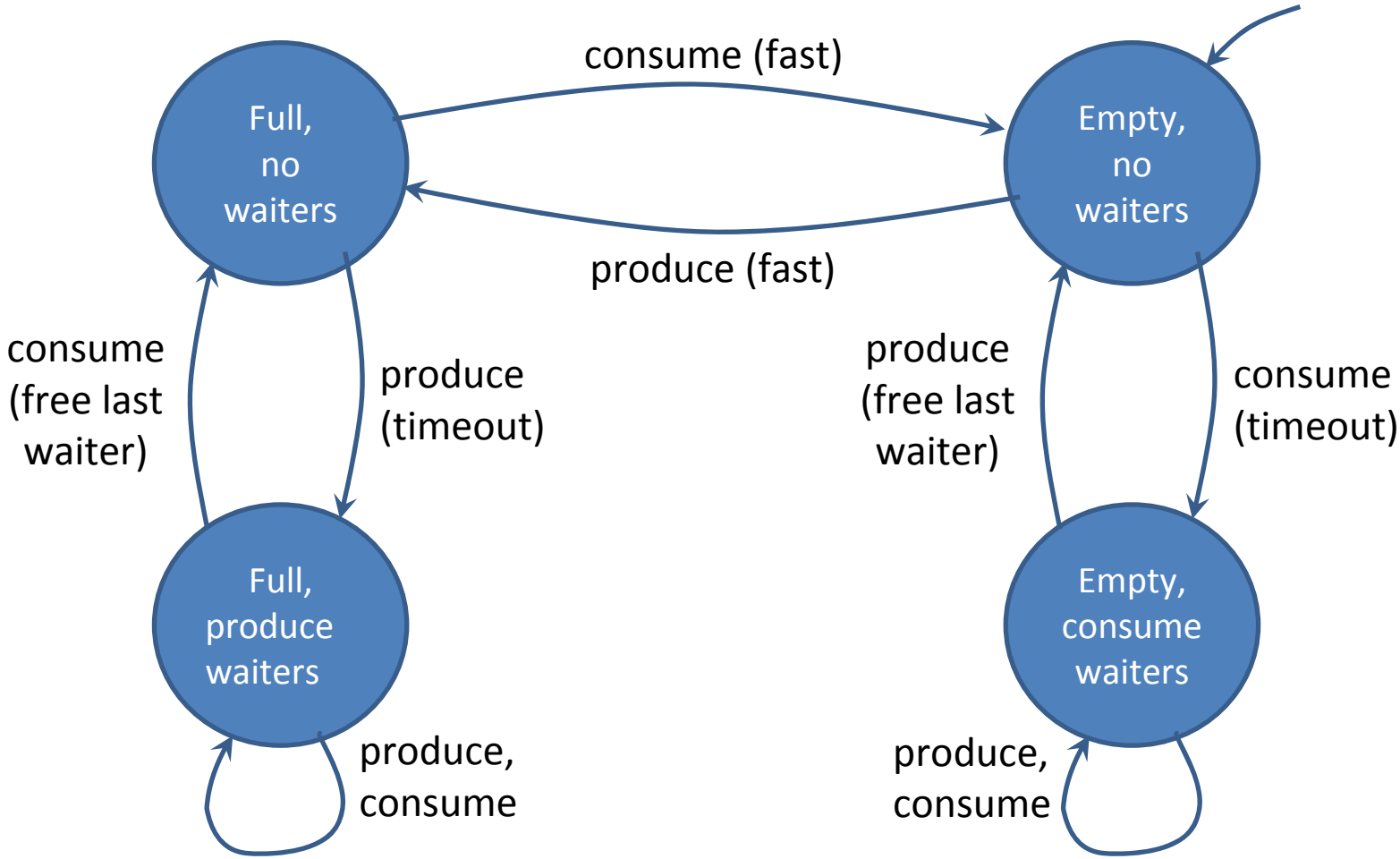
```
void backsub(int n,double b[],
                  double R[][],double x[]){
  volatile SAvar double xs[n];
  forall(i = n-1; i >= 0; i--){
    double ac = b[i];
    for(j = n-1; j > i; j--){
      ac -= R[i][j]*xs[j];
    }
    xs[i] = ac/R[i][i];
    x[i] = xs[i];
  }
}
```



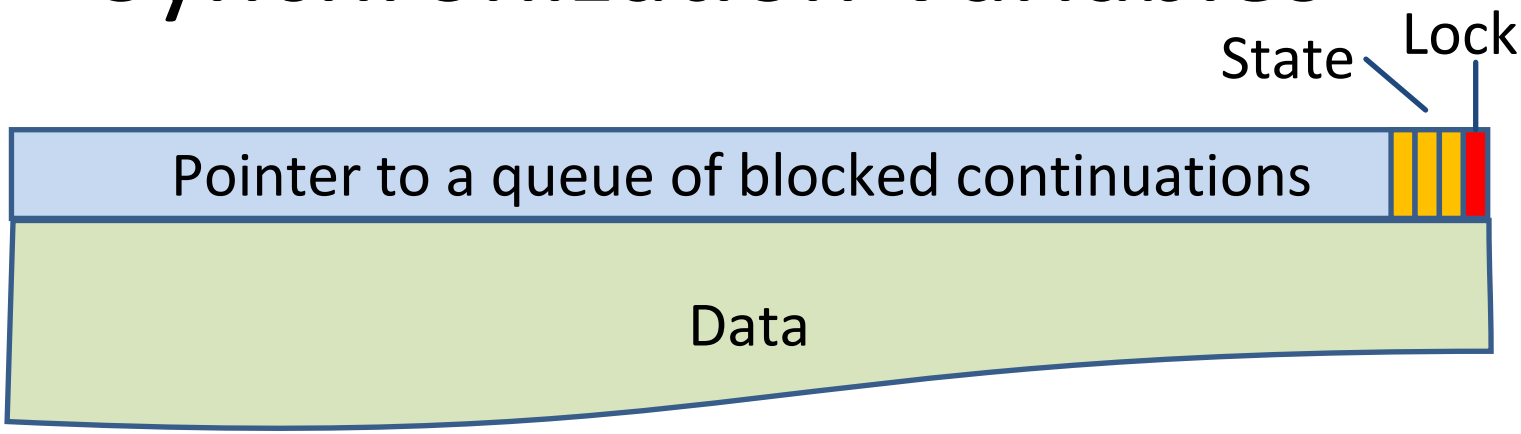
# Single Assignment FSM



# Producer-Consumer FSM



# Synchronization Variables




**`sl = mwaitc(syncvar,state_spec,time_limit)`**

- **`syncvar`** is the address of the synchronization variable
- **`state_spec`** is a bit table defining awaited unlocked States
- **`time_limit`** specifies when **`mwaitc`** will timeout
- **`mwaitc`** returns when Lock is successfully set or on timeout
  - Lock is set if and only if **`state_spec`**[State] is true
- When **`mwaitc`** returns, **`sl`** is State and Lock concatenated

# Consume Example

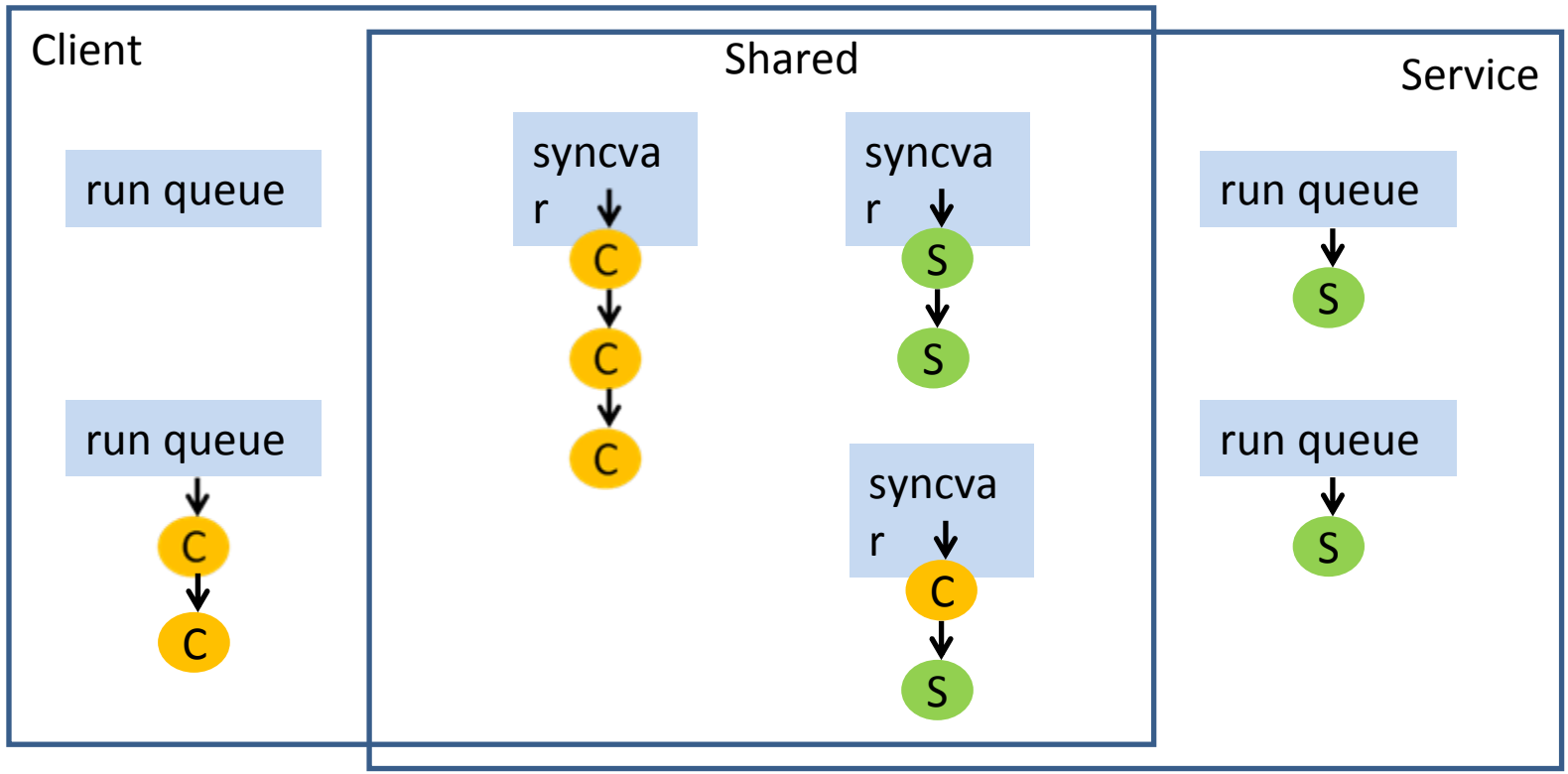
```
int sl = mwaitc(sv,FULLN+EMPTYW+FULLW,TIMEOUT);
switch(sl){
case FULLN:         //fast path
  data = sv->data;
  sv->hdr += -FULLN-1+EMPTYN; //empty and unlock
  return data;
case EMPTYW:
  /*enqueue on *sv and unlock; acquire new work*/
case FULLW:
  /*make enqueued producer runnable; swap sv->data
  with producer data; unlock sv->hdr; return data*/
default:              //timed out
  /*enqueue on *sv; acquire new work*/
}
```

# Implementing **`mwaitc()`**

- A hardware thread waits at one **`syncvar`**
- If a core has several hardware threads it can keep issuing instructions from the others
- In a cache-coherent situation, **`mwaitc`** might monitor L1 for invalidates, then reload the line
- In an incoherent situation, **`mwaitc`** might read occasionally and then try a Compare&Swap
  - If access is cache coherent at some remote site, hardware located there can implement waiting

# Services and Partial Sharing

- Services and clients can share regions of memory for buffering and synchronization
- The main difficulty is protection



# Isolating Services and Clients

- Blocked task state is kept in unshared memory
- Pointers to blocked task state are encrypted
- When a task unblocks, its encrypted pointer is moved to an “indirect queue” in shared space
  - Client and service have distinct indirect queues
- Daemons decrypt and validate the pointers and then re-enqueue them appropriately
  - One daemon per client and one per service
- Daemon tasks can circulate on work queues

# Partitioned Global Address Space

- Two-dimensional addresses: <node, offset>
- Memory access is strongly non-uniform
- Tasks have hard affinity to nodes
  - Closures may be shipped remotely, however
- Support for efficient remote enqueueing and dequeueing would be helpful
  - A complex case: a task in node C unblocks a task affinitized to node A from a syncvar in node B
- Daemons may also be useful in this situation

# Conclusions

- Fine-grain communication among cores is an important but neglected opportunity
- Blocking synchronization plays a key role
- The main trick is to make blocking inexpensive
- Better application scaling will be the result
  - for both client and exascale systems