

Programming Shared Memory Multiprocessors with Deterministic Message-Passing Concurrency: Compiling SHIM to Pthreads

Stephen A. Edwards, Nalini Vasudevan
Columbia University

Olivier Tardieu
IBM

Main Points

- Scheduling-independent message passing works for parallel programming

We use the SHIM language

- This paradigm helps to safely explore schedules

Compiler catches race-related bugs

- Our compiler generates efficient pthreads (C) code

Synthesizing communication the trick

- Results: 3.05 and 3.3 \times speedups on a four-core

JPEG and FFT examples

A SHIM example

Five functions that call each other and communicate through channel *A*

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

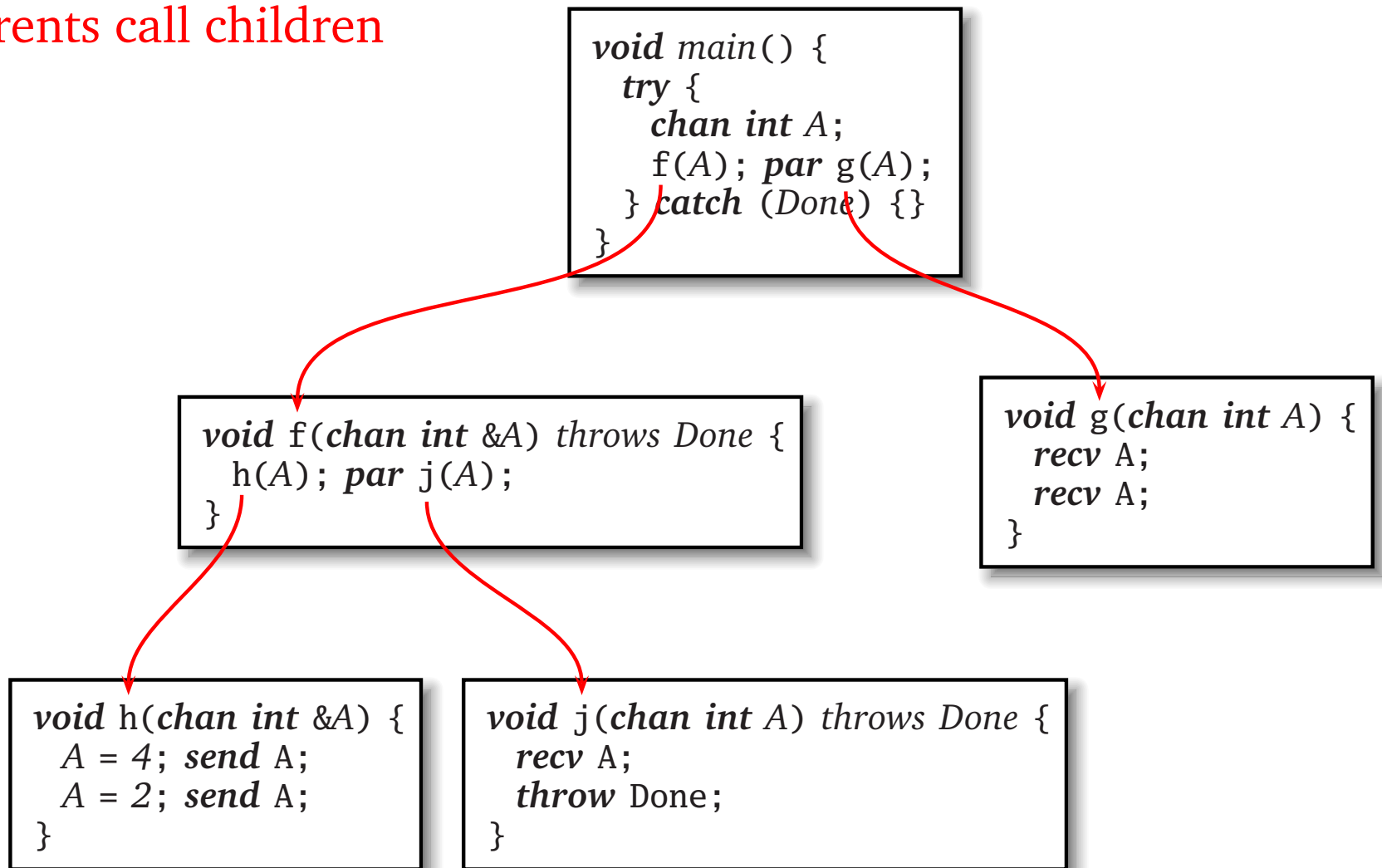
```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

A SHIM example

Parents call children



A SHIM example

h sends 4 on *A*,
g and *j* rendezvous

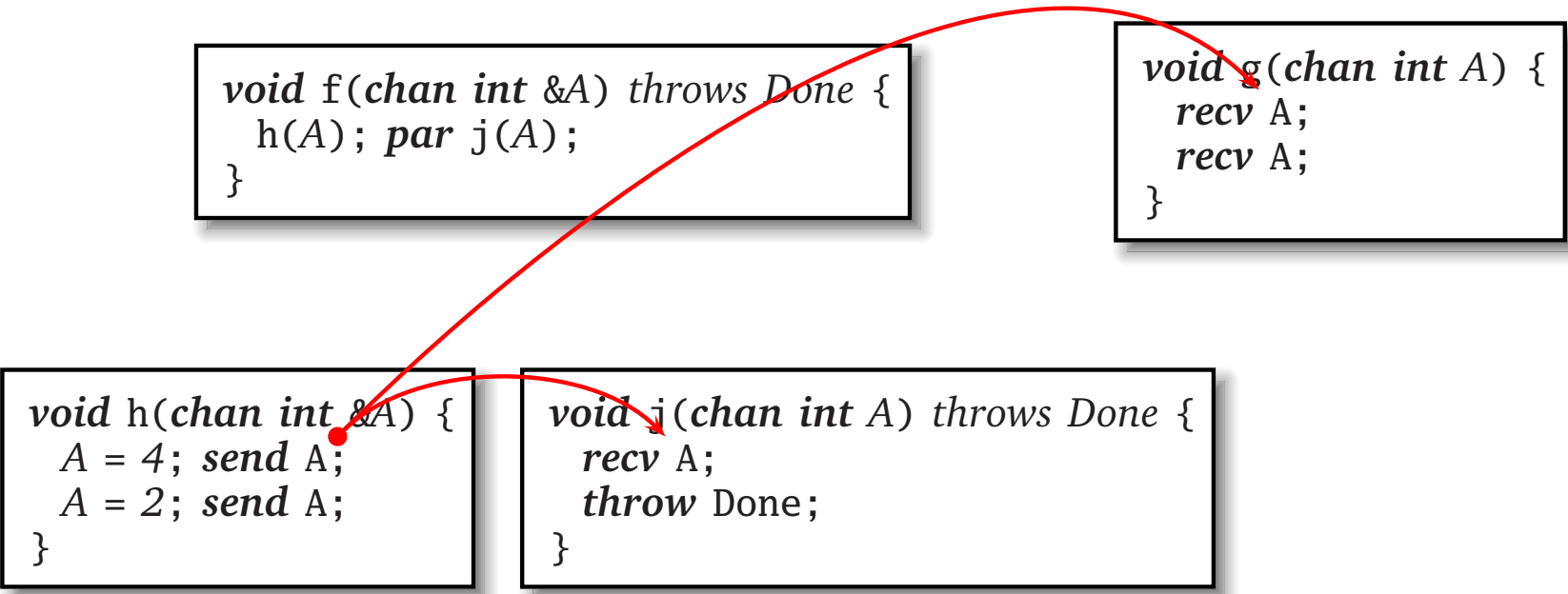
```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```



A SHIM example

j throws an exception.
g and *h* poisoned by
attempting
communication

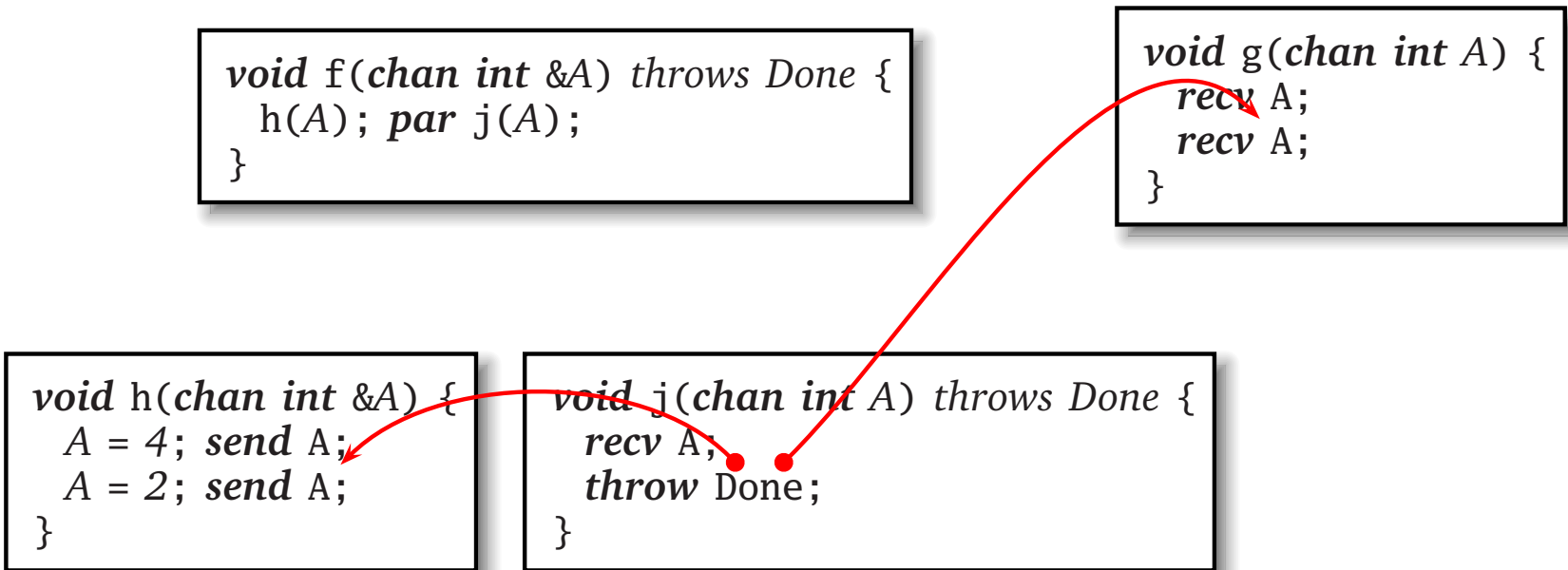
```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```



A SHIM example

Concurrent processes
terminate, control
passed to exception
handler

```
void main() {  
  try {  
    chan int A;  
    f(A); par g(A);  
  } catch (Done) {}  
}
```

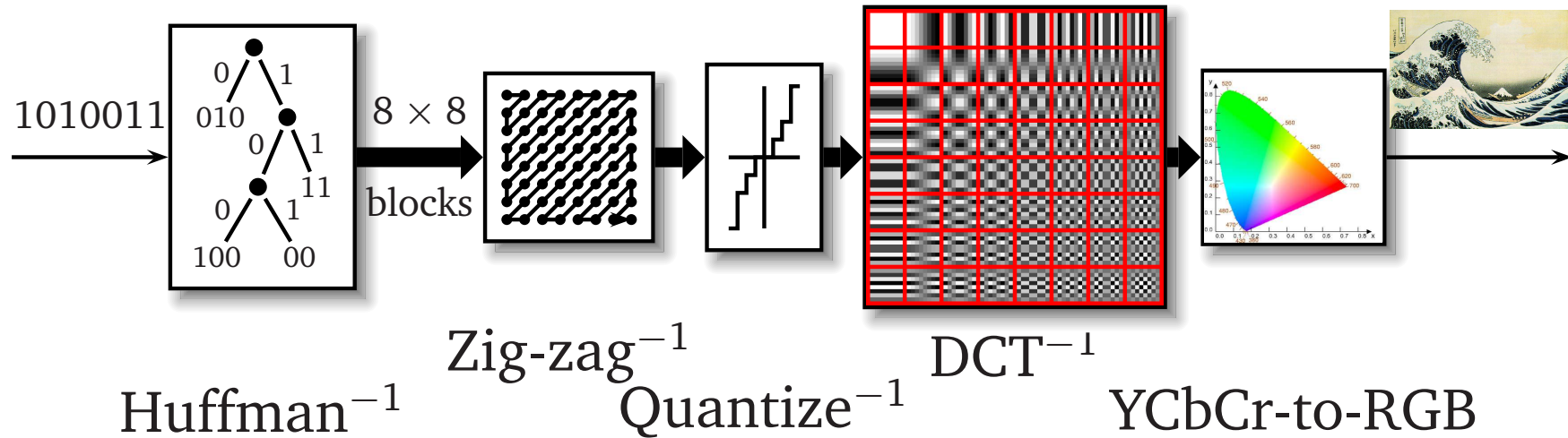
```
void f(chan int &A) throws Done {  
  h(A); par j(A);  
}
```

```
void g(chan int A) {  
  recv A;  
  recv A;  
}
```

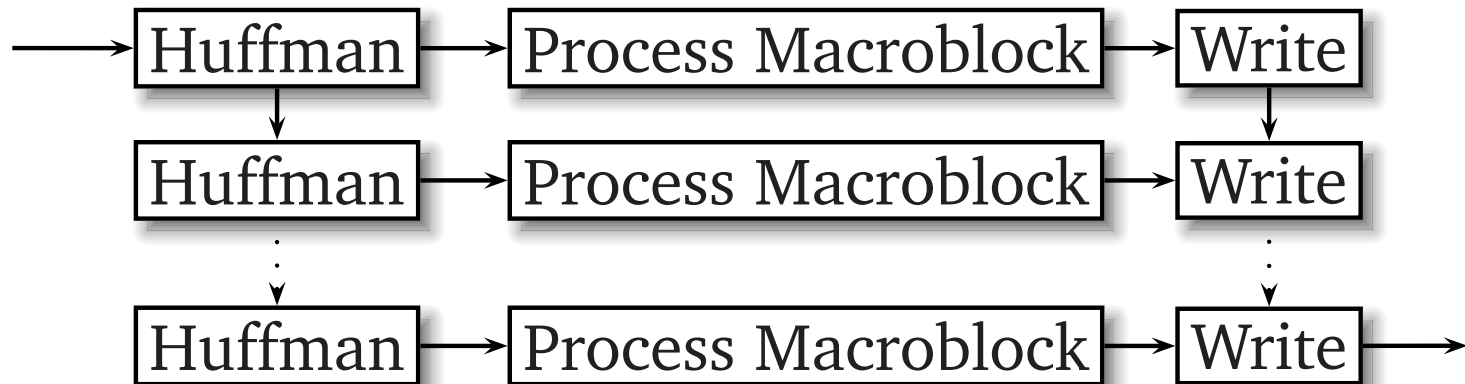
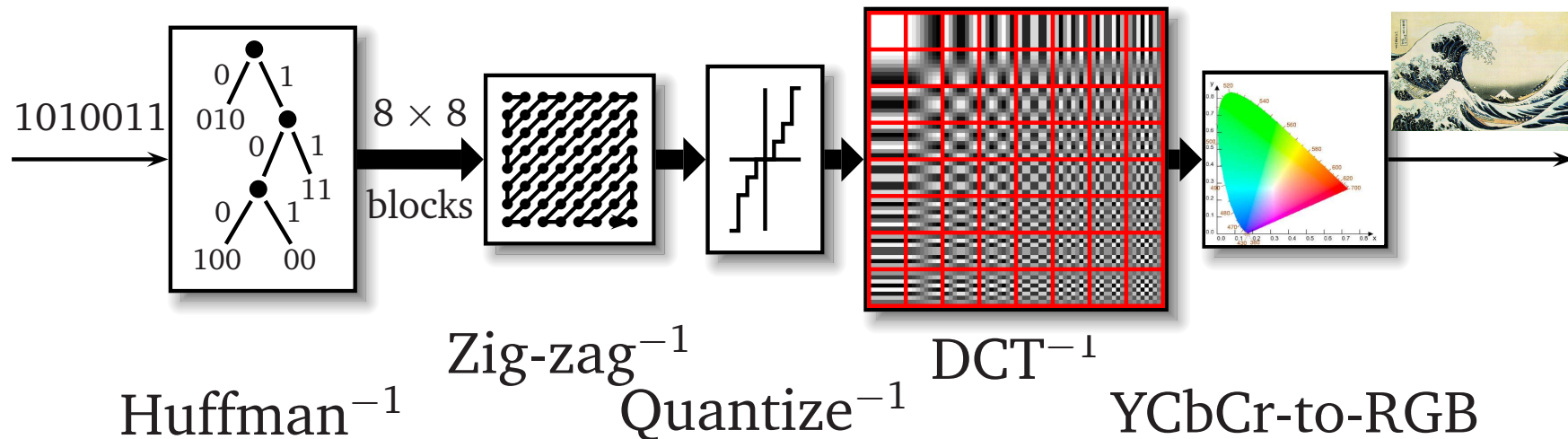
```
void h(chan int &A) {  
  A = 4; send A;  
  A = 2; send A;  
}
```

```
void j(chan int A) throws Done {  
  recv A;  
  throw Done;  
}
```

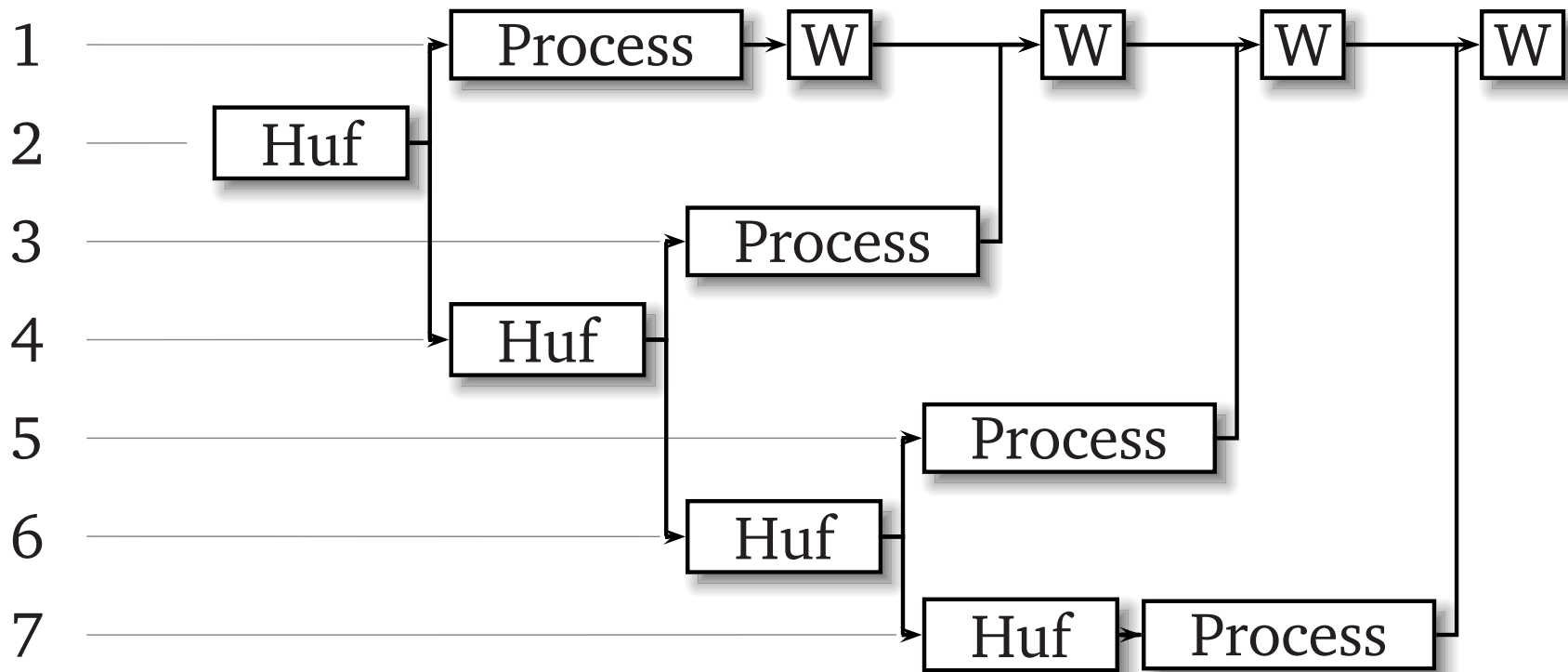
JPEG Decoding



JPEG Decoding



Seven-task JPEG schedule



Idea: minimize communication events

SHIM for the Seven-task Schedule

```
unpacker_state ustate;
```

```
writer_state wstate;
```

```
stripe stripe1, stripe2, stripe3, stripe4;
```

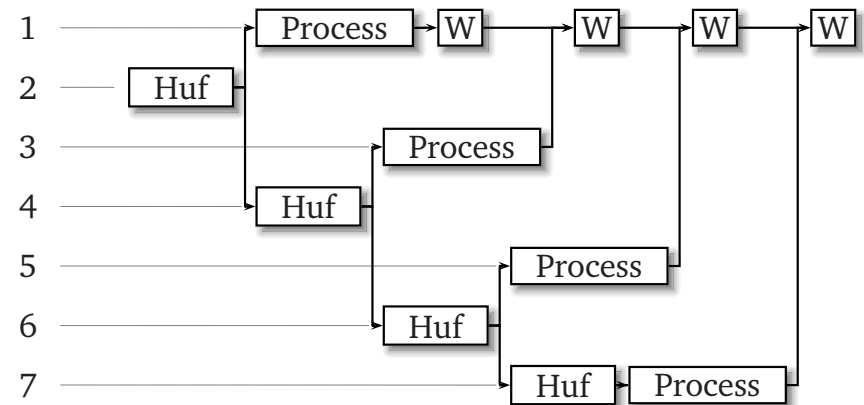
```
pixels pixels1; // to writer
```

```
chan pixels pixels2, pixels3, pixels4;
```

```
void unpack(unpacker_state &state, stripe &stripe) { ... } // Huffman Decode
```

```
void process(const stripe &stripe, pixels &pixels) { ... } // IDCT, etc.
```

```
void write(writer_state &wstate, const pixels &pixels) { ... } // Write to file
```

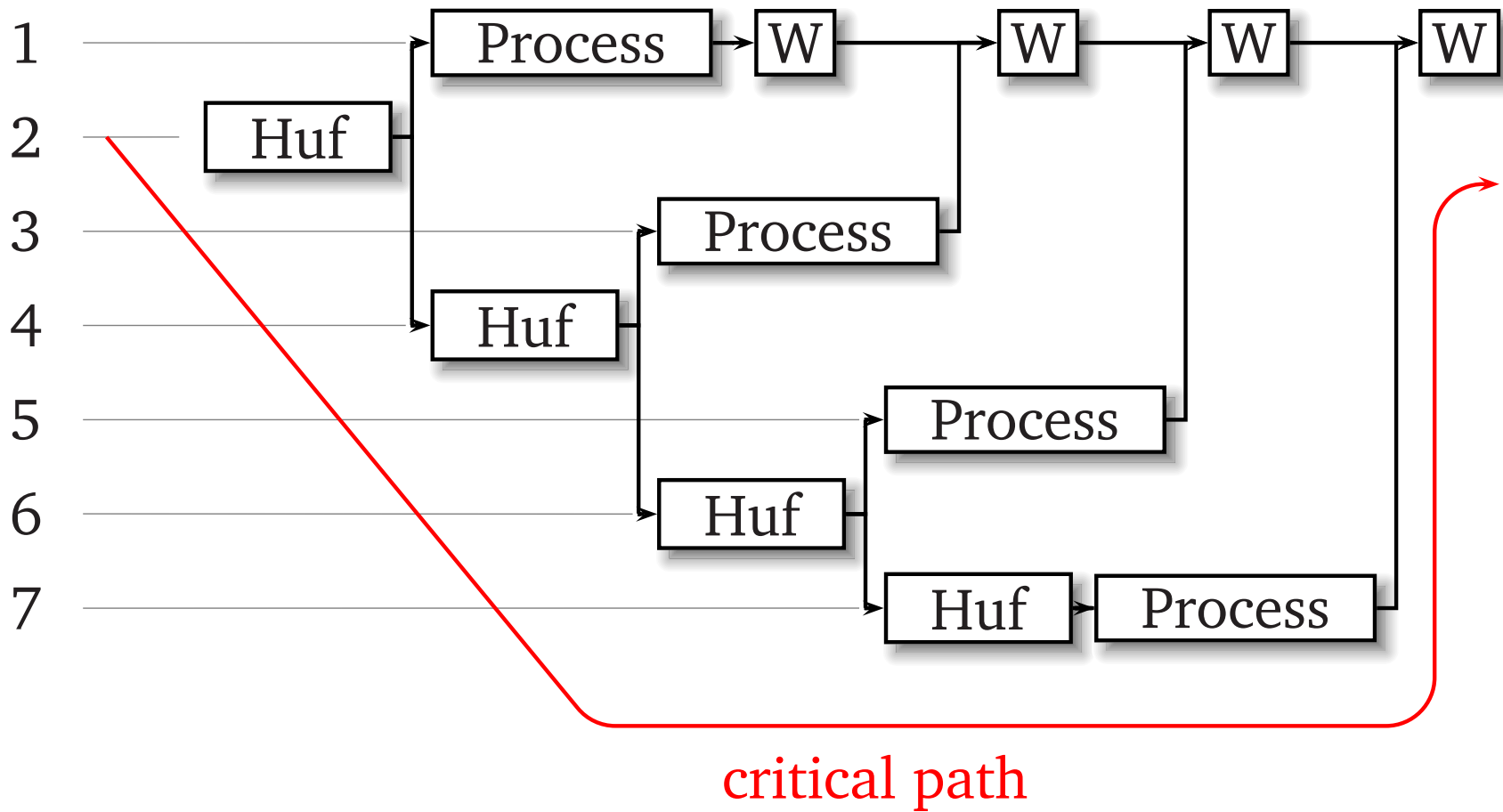


SHIM Enforces Dependencies

```
unpack(ustate, stripe1);
{
  process(stripe1, pixels1); write(wstate, pixels1);
  rcv pixels2; write(wstate, pixels2);
  rcv pixels3; write(wstate, pixels3);
  rcv pixels4; write(wstate, pixels4);
} par {
  unpack(ustate, stripe2);
  {
    process(stripe2, pixels2); send pixels2;
  } par {
    unpack(ustate, stripe3);
    {
      process(stripe3, pixels3); send pixels3;
    } par {
      unpack(ustate, stripe4);
      process(stripe4, pixels4); send pixels4;
    } } }
} } }
```

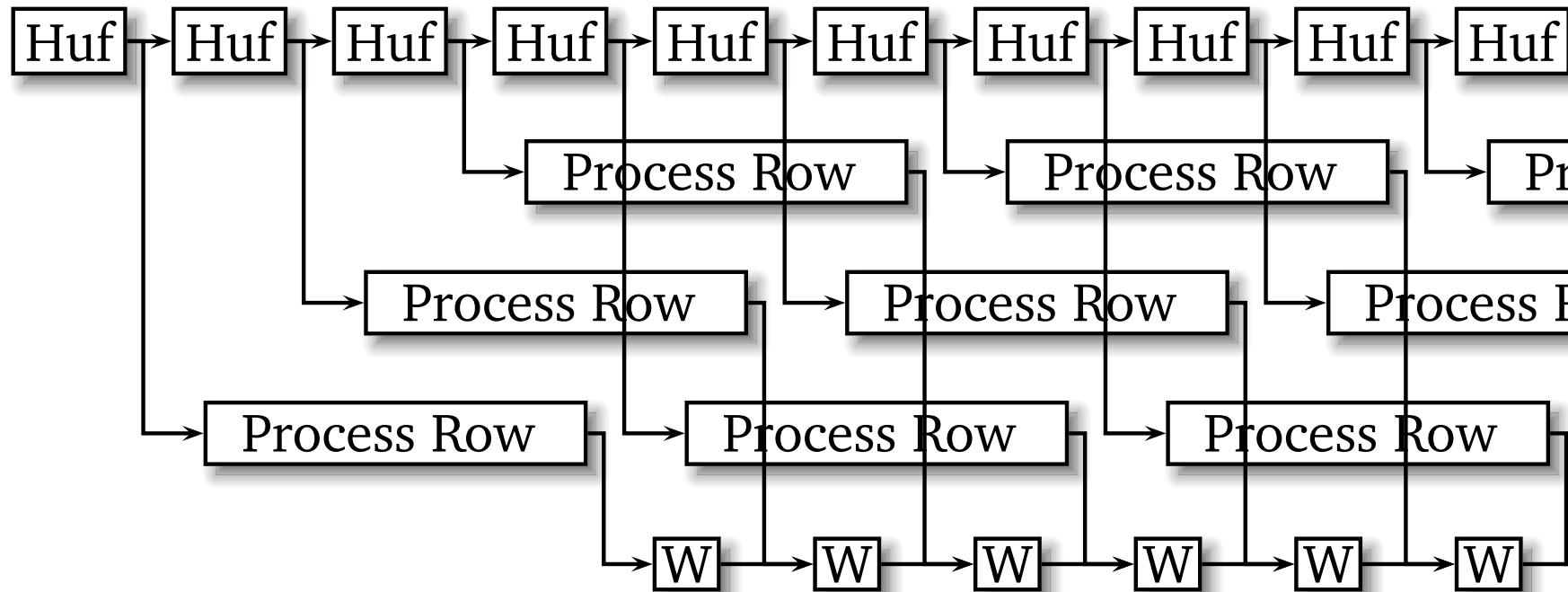
- Writer state local to one process
- Unpacker state can only be passed by reference once
- Trying to run *unpack* or *write* in parallel gives compiler error

Oops



Only achieved a $1.8\times$ speedup

Pipelined JPEG

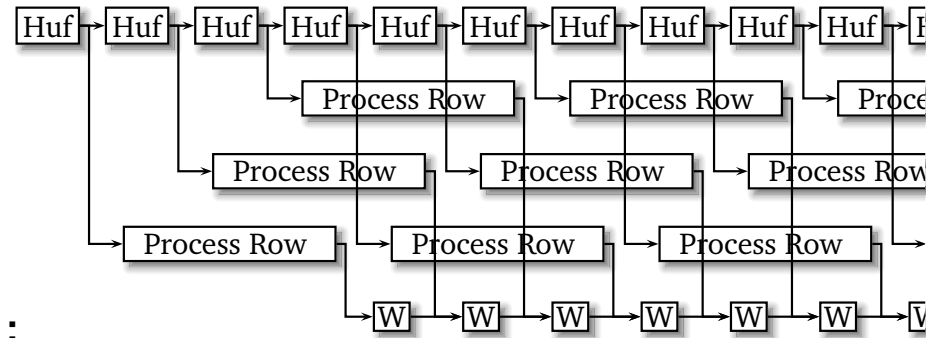


Process a row of blocks at a time (e.g., 64).

Reduce communication; accelerate start-up and termination.

SHIM for Pipelined JPEG

```
try {  
  {  
    for (;;) {  
      unpack(ustate, row1); send row1; if (--rows == 0) break;  
      unpack(ustate, row2); send row2; if (--rows == 0) break;  
      unpack(ustate, row3); send row3; if (--rows == 0) break;  
    } throw Done;  
  } par  
    process(row1, pixels1); par  
    process(row2, pixels2); par  
    process(row3, pixels3); par  
  {  
    for (;;) {  
      recv pixels1; write(wstate, pixels1);  
      recv pixels2; write(wstate, pixels2);  
      recv pixels3; write(wstate, pixels3);  
    } }  
} catch (Done) {}
```









Task and Channel Structures

```
void foo(int a, int a)
{
    chan int c;
}
```









Task and Channel Structures








```
void foo(int a, int a)
{
  chan int c;
}
```

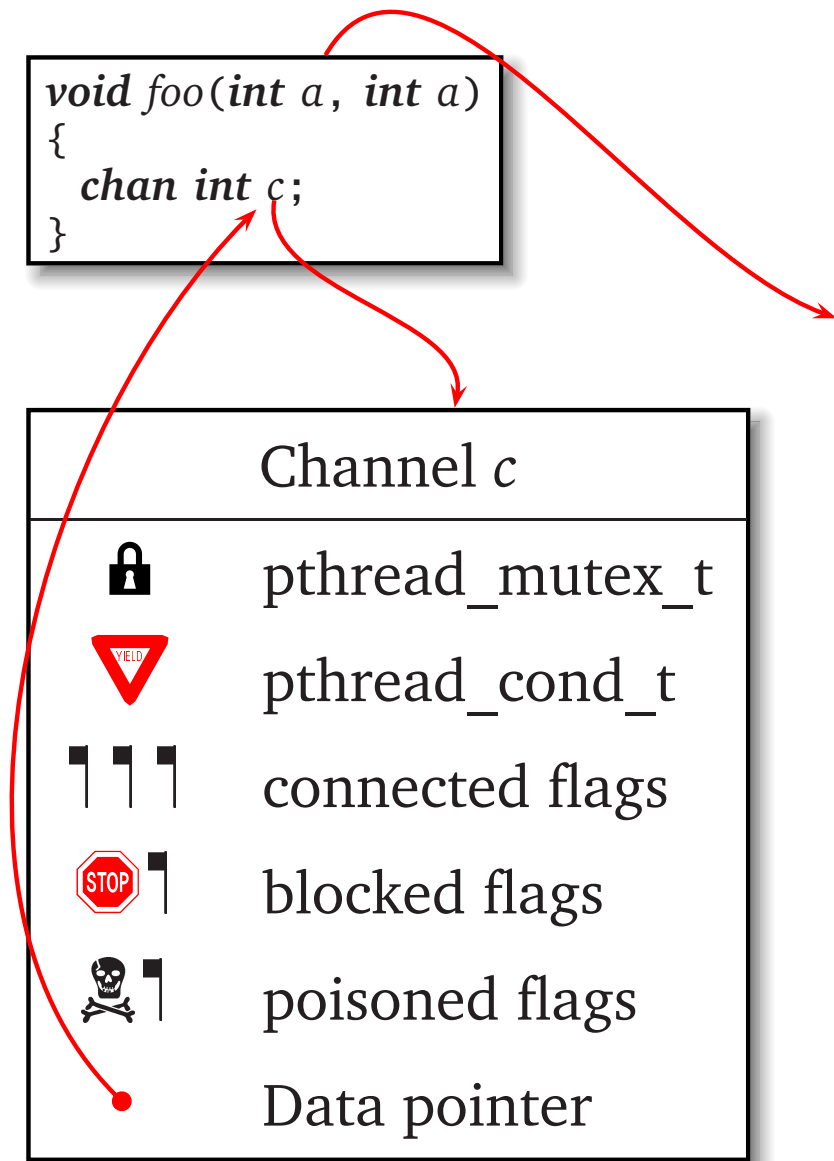
Task <i>foo</i>	
}	pthread_t
	pthread_mutex_t
	pthread_cond_t
 ,  , 	enum State
	# attached children
<i>a</i>	Formal arg.
<i>b</i>	Formal arg.

Task and Channel Structures

```
void foo(int a, int a)
{
  chan int c;
}
```

Channel c	
	pthread_mutex_t
	pthread_cond_t
	connected flags
 	blocked flags
 	poisoned flags
	Data pointer









Task foo	
	pthread_t
	pthread_mutex_t
	pthread_cond_t
 ,  , 	enum State
	# attached children
<i>a</i>	Formal arg.
<i>b</i>	Formal arg.









Task and Channel Structures

```
void foo(int a, int a)
{
  chan int c;
}
```

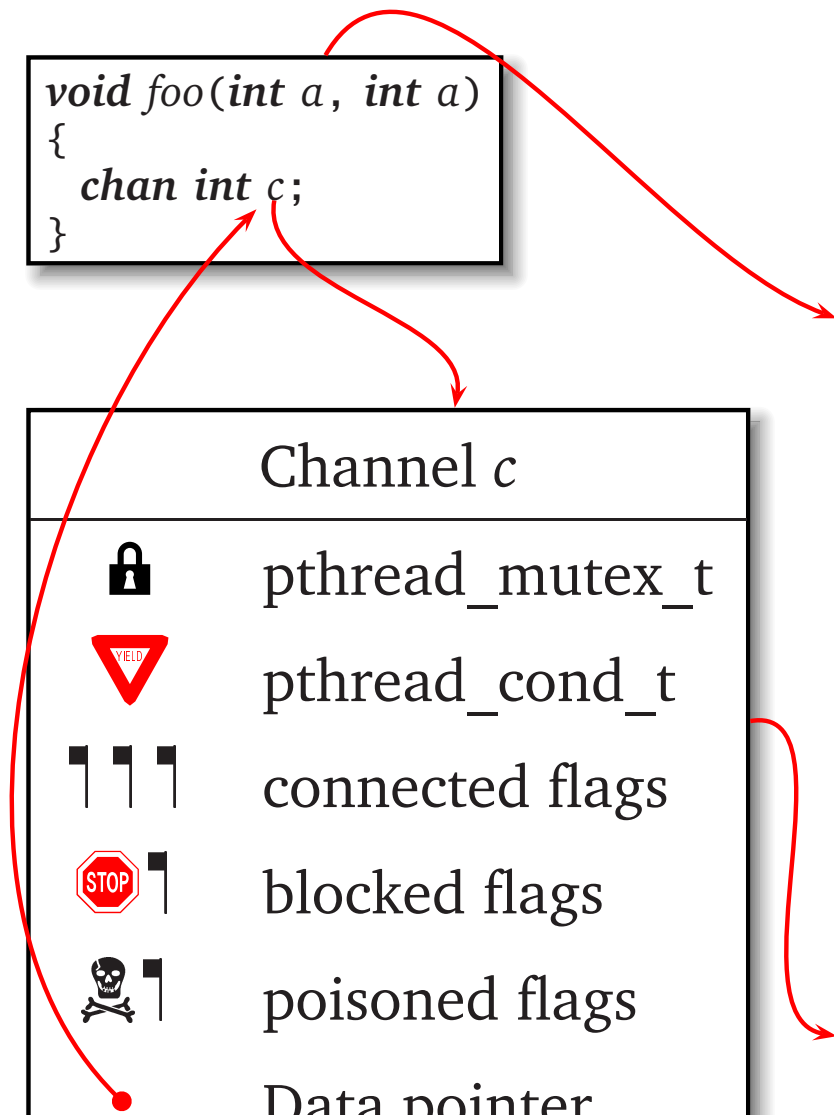
Channel c

	pthread_mutex_t
	pthread_cond_t
	connected flags
 	blocked flags
 	poisoned flags
	Data pointer

Task foo

}	pthread_t
	pthread_mutex_t
	pthread_cond_t
 ,  , 	enum State
	# attached children
a	Formal arg.
b	Formal arg.

```
void event_c() {
  if (c.connected == c.blocked) {
    // Communicate
  } else if (c.poisoned) {
    // Propagate exceptions
  }
}
```



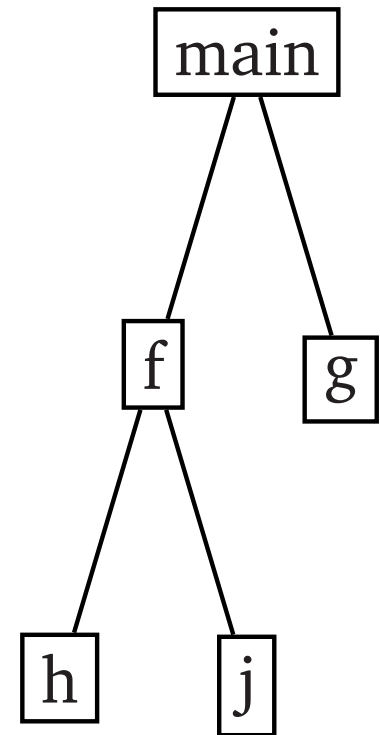
Code for *send* A in h()

```
pthread_mutex_lock(A.mutex);           // Lock for channel A

A.blocked |= (A_h|A_f|A_main);         // Block ancestors, too.
event_A();                             // Communicate if possible

while (A.blocked & A_h) {              // Are we ready?
  if (A.poisoned & A_h) {              // Were we poisoned?
    pthread_mutex_unlock(A.mutex);
    goto _poisoned;                    // Handle exception
  }
  pthread_cond_wait(A.cond, A.mutex);   // Yield
}

pthread_mutex_unlock(A.mutex);
```



An Event Function

```
void event_A() {
    unsigned int can_die = 0, kill = 0; // Flags
    if (A.connected == A.blocked) { // Communicate
        A.blocked = 0; // Unblock everybody
        if (A.connected & A_g) *A.g = *A.main; // Copy data
        if (A.connected & A_j) *A.j = *A.main;
        pthread_cond_broadcast(A.cond); // Awaken blocked tasks
    } else if (A.poisoned) { // Propagate exceptions
        can_die = blocked & (A_g|A_h|A_j); // Compute can_die
        if (can_die & (A_h|A_j) == A.connected & (A_h|A_j)) can_die |= blocked & A_f;
        if (A.poisoned & (A_f|A_g)) { // Compute kill
            kill |= A_g; if (can_die & A_f) kill |= (A_f|A_h|A_j);
        }
        if (A.poisoned & (A_h|A_j)) { kill |= A_h; kill |= A_j; }
        if (kill &= can_die & ~A.poisoned) { // Anybody to poison?
            pthread_mutex_unlock(A.mutex);
            if (kill & A_g) { // Poison g if necessary
                pthread_mutex_lock(g.mutex);
                g.state = POISON;
                pthread_mutex_unlock(g.mutex); }
            // also poison f, h, and j if in kill set...
            pthread_mutex_lock(A.mutex);
            A.poisoned |= kill; pthread_cond_broadcast(A.cond);
        }
    }
}
```

Skeleton for Task f

restart:

Wait until my parent sets
my state to RUN

Body of the task

terminated:

Disconnect from each of
my channels

Set my state to STOP

goto detach

poisoned:

...

deatch:

Tell my parent that I have
detached

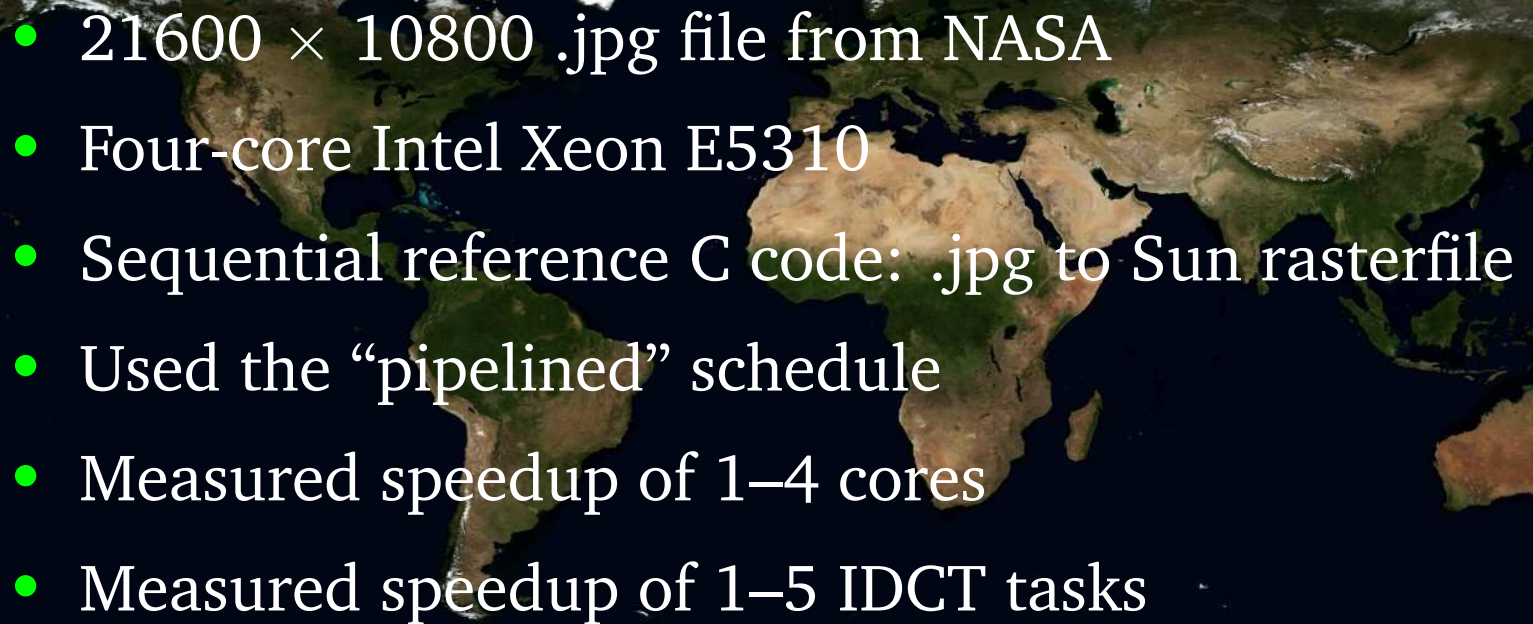
goto restart

Skeleton for Task f

```
void *_thread_f(void *_ignored) {  
    int *A; // Actual argument  
  
    _restart: // Wait for RUN  
    pthread_mutex_lock(f.mutex);  
    while (f.state != RUN)  
        pthread_cond_wait(f.cond, f.mutex);  
    A = f.A;  
    pthread_mutex_unlock(f.mutex);  
  
    // body of f() ...
```

```
    _terminated: // Disconnect f from channel A  
        pthread_mutex_lock(A.mutex);  
        A.connected &= ~A_f; event_A();  
        pthread_mutex_unlock(A.mutex);  
  
        pthread_mutex_lock(f.mutex); // State to STOP  
        f.state = STOP;  
        pthread_mutex_unlock(f.mutex);  
        goto _detach;  
  
    _poisoned: // Handle poisoning  
        // ...  
  
    _detach: // Detach from parent (main)  
        pthread_mutex_lock(main.mutex);  
        --main.attached_children;  
        pthread_cond_broadcast(main.cond);  
        pthread_mutex_unlock(main.mutex);  
        goto _restart;  
}
```

JPEG Experiment

- 
- 21600 × 10800 .jpg file from NASA
 - Four-core Intel Xeon E5310
 - Sequential reference C code: .jpg to Sun rasterfile
 - Used the “pipelined” schedule
 - Measured speedup of 1–4 cores
 - Measured speedup of 1–5 IDCT tasks

JPEG Results

Cores	Tasks	Time	Total	Total/Time	Speedup
1	1	25s	20s	0.8	1.0× (def)
1	1+3+1	24	24	1.0	1.04
2	1+3+1	13	24	1.8	1.9
3	1+3+1	11	24	2.2	2.3
4	1+3+1	8.7	25	2.9	2.9
4	1+1+1	16	24	1.5	1.6
4	1+2+1	9.3	25	2.7	2.7
4	1+3+1	8.7	25	2.9	2.9
4	1+4+1	8.2	25	3.05	3.05
4	1+5+1	8.6	25	2.9	2.9

FFT Experiment (testing roundoff)

40 MB .wav file (16-bit stereo)



1024-point 4.28 fixed-point FFT



inverse FFT



.wav file

- Same hardware as JPEG (Xeon Quad-core)
- Baseline: sequential C from *Numerical Recipes*
- 1–4 cores, “pipelined” with 1 1024-sample block
- 1–4 cores, “pipelined” with 16 1024-sample blocks

FFT Results

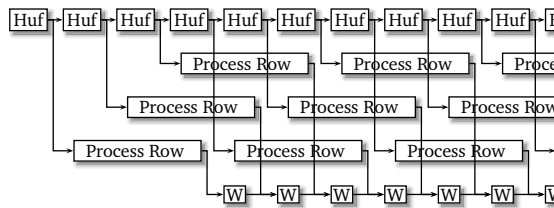
Code	Cores	Time	Total	Total/Time	Speedup
Handwritten C	1	2.0s	2.0s	1.0	1.0× (def)
Sequential SHIM	1	2.1	2.1	1.0	0.95
Parallel SHIM	1	2.1	2.1	1.0	0.95
Parallel SHIM	2	1.3	2.0	1.5	1.5
Parallel SHIM	3	0.92	2.1	2.2	2.2
Parallel SHIM	4	0.86	2.1	2.4	2.3
Parallel 16	1	1.9	1.9	1.0	1.1
Parallel 16	2	1.0	1.9	1.9	2.0
Parallel 16	3	0.88	1.9	2.1	2.2
Parallel 16	4	0.6	1.9	3.2	3.3

Conclusions

- Scheduling-independent message passing language

SHIM

- Exploring schedules interesting, safe



- Our compiler generates C code with pthreads calls



- Efficient: 3.05 and $3.3\times$ speedups on a four-core

