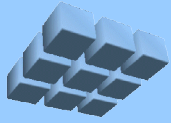


# **Configurable Multiprocessing: An FIR Filter Example**

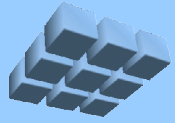
Cmpware, Inc.



# Introduction

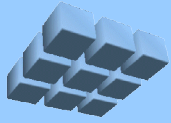
- Multiple processors on a device common
- Thousands of 32-bit RISC CPUs possible
- Advantages in:
  - Performance
  - Power consumption
  - Programmability

**Q:** How best to program such devices?



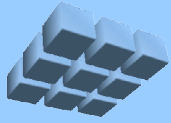
# Configurable Microprocessing

- **Multiple standard processor cores**
- **Fast point to point interconnection**
- **Balanced communication / computation ratio (1:1 or better)**
- **Use standard compilers / tools**
- **Use *Cmpware* to design and program the multiprocessor**



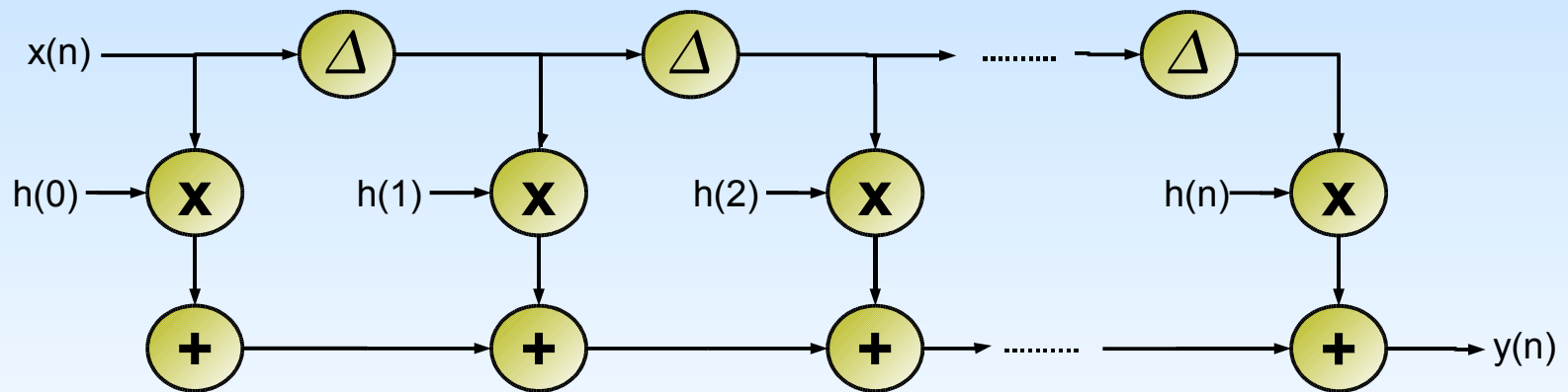
# The *Cmpware* Software

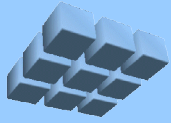
- Uses standard compilers / assemblers / etc.
- Memory Mapped IO for communication
  - Does not break compilers and other tools
  - Does not break processor IP
  - Provides fast, high-bandwidth communication
  - Provides a simple programming model
- *Eclipse* based IDE
  - Multiprocessor simulation
  - Multiprocessor state and performance display



# Example: FIR Filter

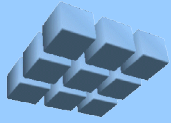
- Finite Impulse Response (FIR) filter
- Digital Signal Processing (DSP) filter
- Often implemented in hardware and software
- Various methods of parallelizing





# The FIR Filter Implementation

- Parameterized on taps and processors
- **FIR()** and **shift()** subroutines standard serial “C” code
- Parallel code constructs at highest level, controlling serial subroutines
- Data read from **east** and sent out to **west**
- Similar structurally to FIR diagrams



# Example: FIR Filter

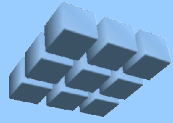
```
void _start(void) {
    int node;
    int ntaps;

    /* Get the parameters */
    node = *west;
    ntaps = *west;

    /* Send the parameters to the next node */
    *east = (node-1);
    *east = ntaps;

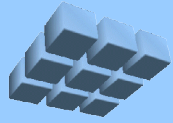
    for (;;) {
        *east = FIR(ntaps, *west);
        *east = shift(ntaps, *west);
    } /* end for() */

} /* end _start() */
```



# Example: FIR Filter (cont.)

```
/*  
** This method computes the FIR.  
**  
** @param ntaps The number of taps in the  
**           FIR filter.  
**  
** @param sum The partial sum into the FIR.  
**  
** @return This method returns the FIR result.  
**/  
  
int FIR(int ntaps, int sum) {  
    int i;  
  
    for (i=0; i<ntaps; i++)  
        sum += h[i] * z[i];  
  
    return (sum);  
  
} /* end FIR() */
```



# Example: FIR Filter (cont.)

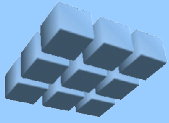
```
int shift(int ntaps, int zIn) {
    int i, zOut;

    /* Save the last value (being shifted out) */
    zOut = z[ntaps-1];

    /* Shift the delay line */
    for (i=ntaps-2; i>=0; i--)
        z[i+1] = z[i];

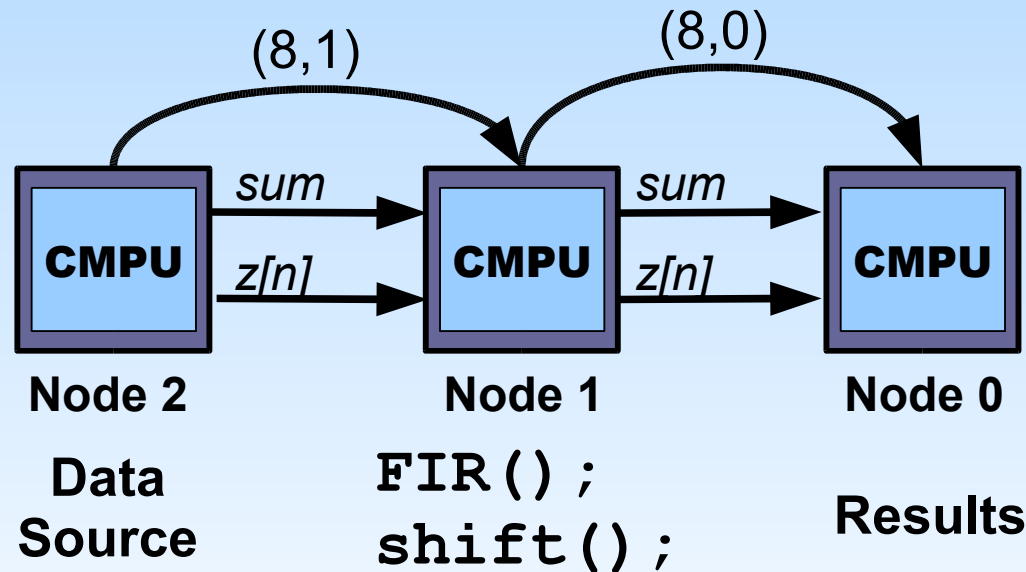
    /* Add in the new shifted in value */
    z[0] = zIn;

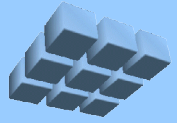
    return (zOut);
} /* end shift() */
```



# FIR Filter Implementation

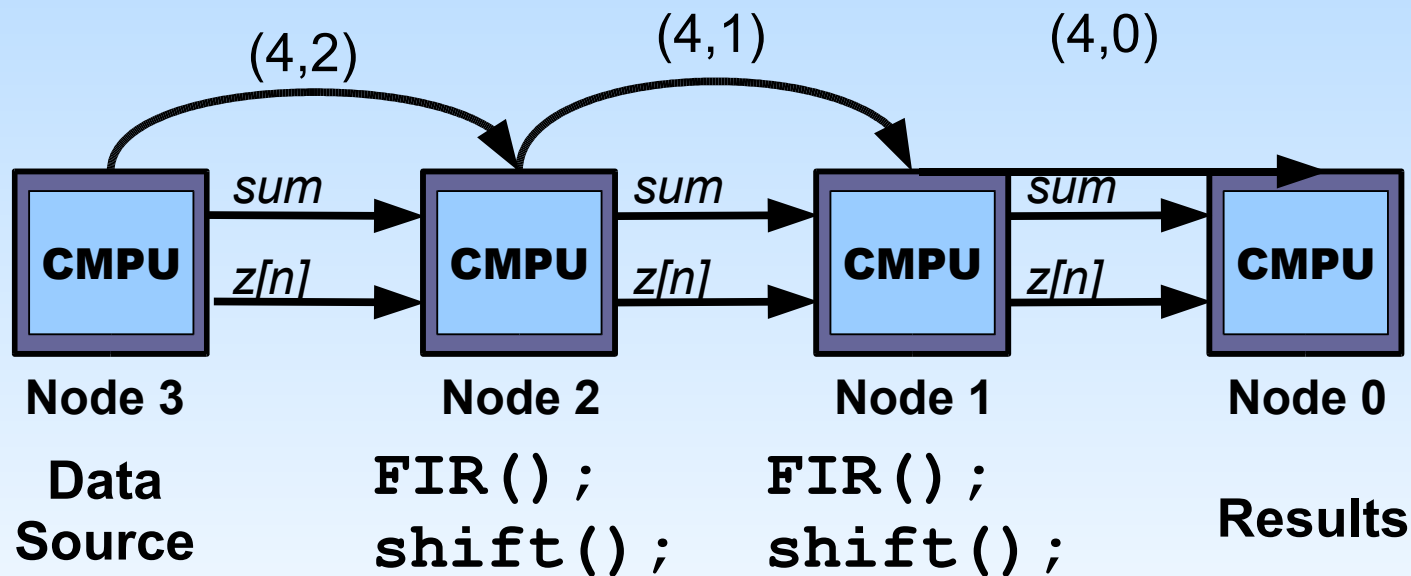
- Three nodes: data source, processor, consumer
- Single 8-tap FIR

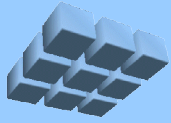




# FIR Filter Implementation (cont.)

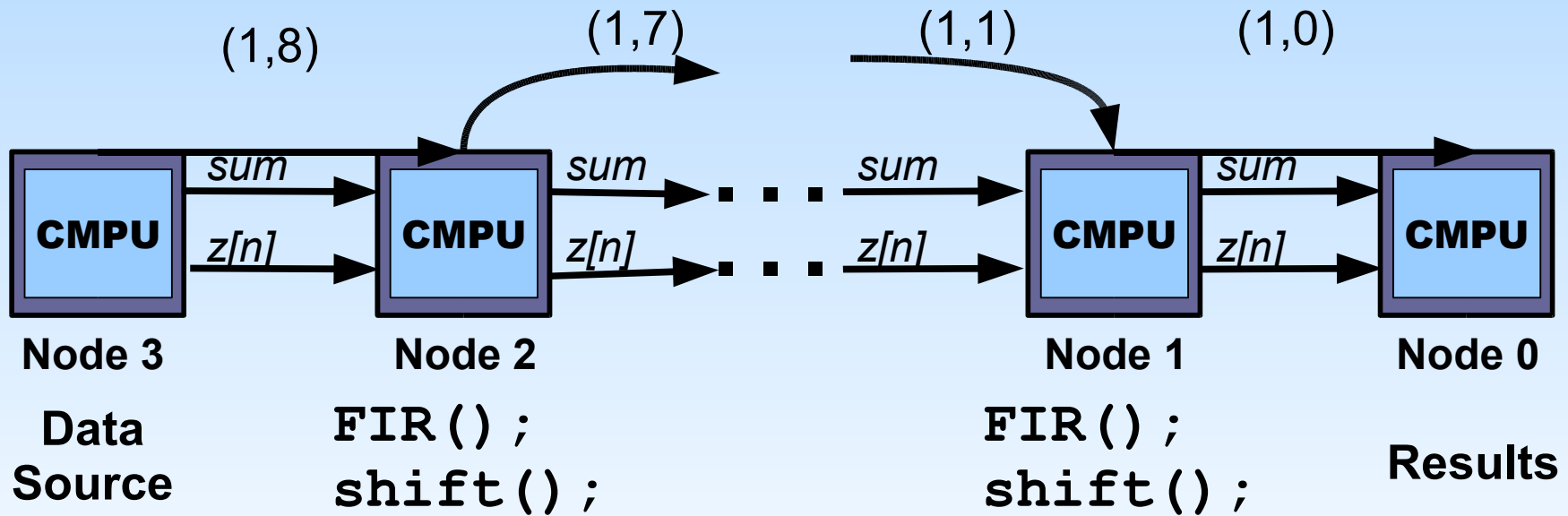
- Four nodes: data source, processor 1, processor 2, consumer
- 4x2 8-tap FIR

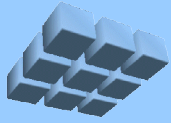




# FIR Filter Implementation (cont.)

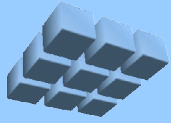
- 8 nodes: data source, 8 processors, consumer
- 8x1 8-tap FIR





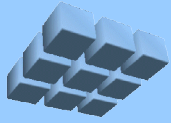
# The *Cmpware* Software

- All standard C
- Same code used for various processor configurations
- Can select number of processors dynamically
- Communication synchronize computation
- Computations only start when inputs are available (pseudo-dataflow)
- Power / performance tradeoffs unavailable in other solutions



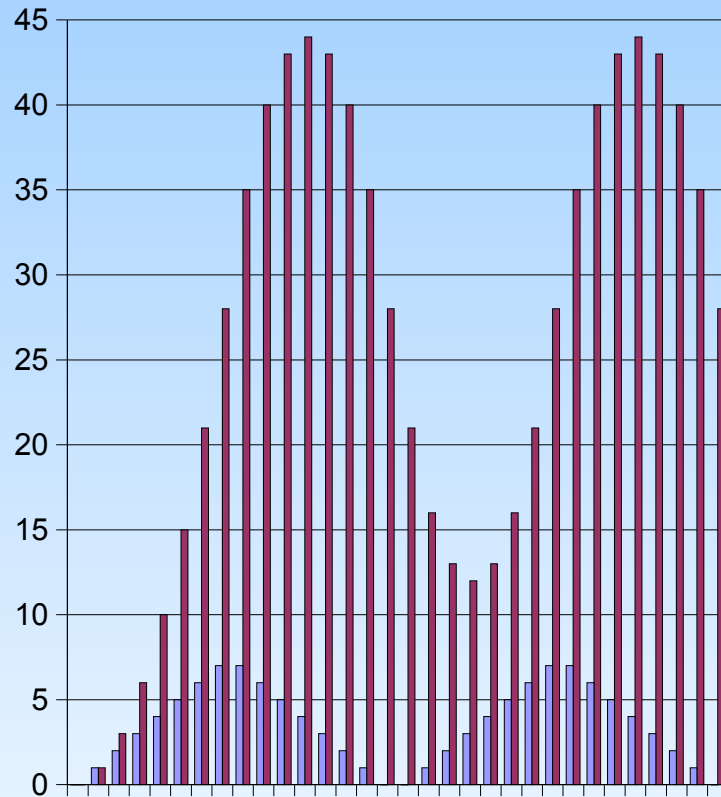
# Synchronization

- FIR filter not simple to parallelize
  - CMP communication synchronizes computation
  - Hardware solution is to:
    - Use slow N-input adder
    - Use pipelined adder tree
    - Add pipeline delay stages
- *All of these HW solutions are very inflexible*

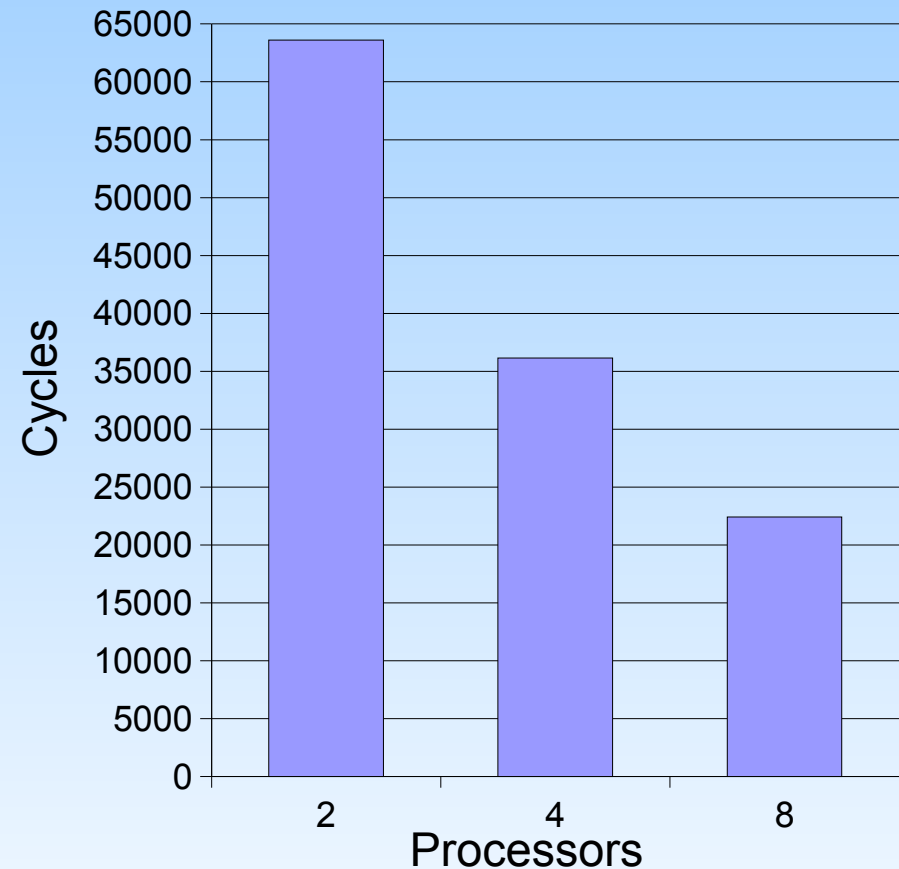


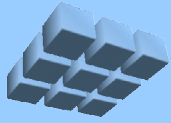
# FIR Filter Speedup

## 8 Tap FIR Filter



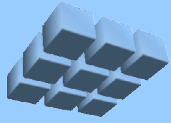
## 8 Tap FIR Filter Speedup





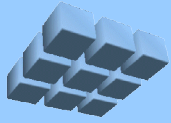
# Using CMP Parallelism

- Simplest parallelism at 'task' (subroutine)
- 1:1 communication / computation permits speedups for very fine grained parallelism
- Parallelism can be exploited at a very low level (if required)
- Parallelism at several levels easily exploited
- Excellent control over performance via level of parallelization



# The Cmpware Tools

- Eclipse IDE based
- Fast, simple access to multiprocessor data
  - Source level trace
  - Memory
  - Registers
  - Performance / profiling, and more ...
- Pluggable CPU simulator / model generator
- Fully extensible API



# The Cmpware Tools

The screenshot displays the Cmpware software interface within the Eclipse Platform. The main window is titled "Cmpware - Welcome - Eclipse Platform" and contains several panes:

- Links Table:** A table with columns for I/O, Address, Value, Count, and Stalls. It lists several memory locations with their respective I/O directions and values.
- CMP Array:** A grid of blue 3D cubes representing the array. One cube in the third row, third column is highlighted with a grey border.
- Status Window:** A small window at the bottom right showing the status of selected processors: P(0,0) selected, P(0,1) selected, P(0,0) selected, and P(2,2) selected.

I/O	Addr.	*	Value	Count	Stalls
In	80000004		0	0	0
In	80000008		0	0	0
In	8000000c		0	0	0
In	80000000		0	0	0
Out	80000004		0	0	0
Out	80000008		0	0	0
Out	8000000c		0	0	0
Out	80000000		0	0	0



# Configurable Multiprocessing from Cmpware, Inc.

