Sorting Using the Xilinx Virtex-4 Field Programmable Gate Arrays on the Cray XD1

Stephen Bique[1], Wendell Anderson[1], Marco Lanzagorta[2], and Robert Rosenberg[1]

[1] *Center for Computational Science*
Naval Research Laboratory
Washington, D.C.
USA

[2] *ITT Industries*
Alexandria, Virginia
USA

This work was performed entirely on the Cray XD1 system at NRL-DC under the auspices of the U. S. Department of Defense (DoD) High Performance Computer Modernization Program (HPCMP).

Outline

- Problem Statement
- Architecture
- Mitrion-C
- Parallel Implementations
  - Pallel Bubblesort
  - Counting Sort
  - Mergesort
  - Novel Combinations of Algorithms
- Summary of Speedups Observed

Why Sorting?

- HPC Applications
- Listed as Target Application [Cray]
- Users: Do there exist examples in Mitrion-C?

Problem Statement

- Sort a large number of lists on FPGA
  - Carry out a stable sort on a list of keys
  - Return a list that contains the locations of the keys after sorting
- Compare the performance using different
  - Types of parallelism
  - Algorithms, and
  - Implementations in Mitrion-C

Example

Given

$$\langle 7,1,6,3,4,5,3,8 \rangle$$

return

$$\langle 1,3,6,4,5,0,7 \rangle$$

How do we measure performance?

- Generate as many pseudorandom permutations as fit into memory on SMP using a good linear congruential generator
- Calculate

$$speedup = \frac{time\ running\ qsort()\ on\ SMP}{time\ running\ FPGA}.$$

Exploit the Architecture

Read and write to external memory

- There are a large number of lists, and
- Speed of I/O operations is vastly asymmetrical

Source	Destination	Read	Write
FPGA	FPGA QDR	Fast (3.2 GB/s)	Fast (3.2 GB/s)
FPGA	Host RAM	Unavailable	Slow (0.7 GB/s)
Host	FPGA QDR	Slow (10 KB/s)	Fast (3.2 GB/s)
Host	Host RAM	Fast (GB/s)	Fast (GB/s)

Hypothesis

- Sorting is already fast on AMD Opteron

$$\Theta(n \log n)$$

- Reading/Writing external QDR memory one word at a time

$$\Omega(n)$$

- Reasonably expect speedup is limited

$$O(\log n)$$

Virtex-4 FPGA

- Four Quad Data Rate (QDR) II SRAM cores
  - 4 × 524288 64-bit words (2M 64-bit)
  - Dual ported (simultaneous reads and writes)
    - Read one word at a time
    - Write one word at a time
- 288 Block RAMs
  - Different configurations (1,2,4,9,18,36-bit)
  - Up to 144K 36-bit, or 288K 18-bit, or 576K 9-bit,…
  - Dual ported
    - Two reads simultaneously
    - A read and a write simultaneously
  - FIFO support

Overview of Mitrion-C

- Execute all statements in parallel!
  - No “PAR” and “SEQ” constructs
  - Exception: data dependency
- Tuples
  - Sequence of different types
  - Good fit for parallel computations
    - Multiple inputs
    - Multiple outputs
- Function “templates”
  - No function calls
  - Pass-by-value

Data and Loop Structures

Collection	for	foreach	while
vector	Pipelined *O(n)*	Parallel *O(1)*	Sequential *O(n)*
list	Sequential *O(n)*	Sequential *O(n)*	Sequential *O(n)*
stream	Sequential *O(n)*	Sequential *O(n)*	Sequential *O(n)*

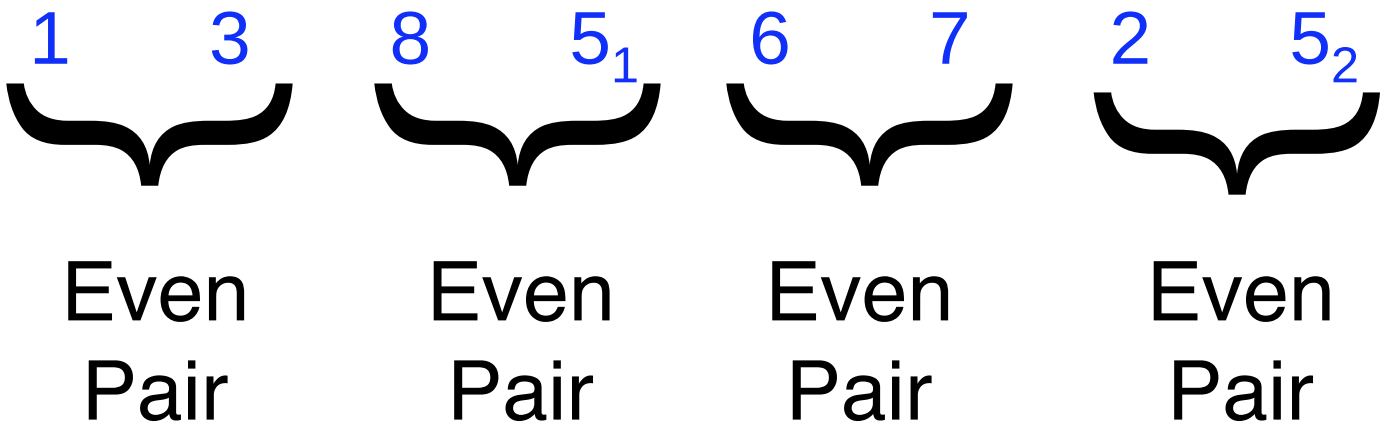
Simulator

- Not clock accurate
  - More steps in simulation might yield faster implementation in hardware!
  - Different number of steps reported in GUI and batch modes
  - Not an accurate hardware simulation
- Reset does not reopen files
- Throughput Analysis Window

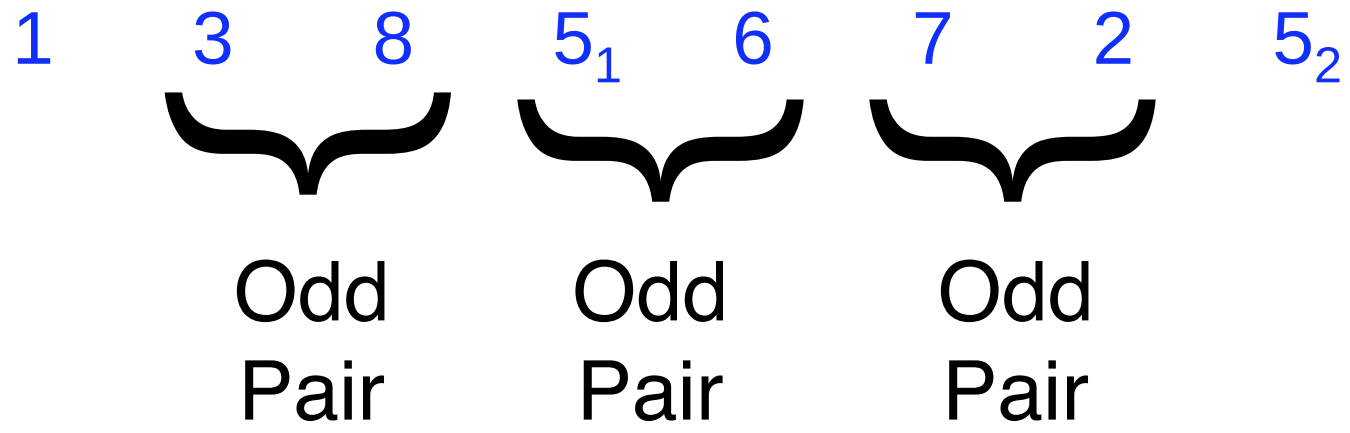
Simulator

- Color scheme for nodes
  - Green - running
  - Grey - waiting
  - Red - halted
- Color scheme for edges
  - Yellow - reversed dependency
  - Blue - connect instance tokens

Definition for Parallel Bubblesort Algorithm



Definition for Parallel Bubblesort Algorithm



Parallel Bubblesort Algorithm

Input: **L** a list
Output: **J** indices for sorted list
Requirement: The length of **L** is even.

1. Sort all even pairs in parallel.
2. Sort all odd pairs in parallel.
3. Repeat steps 1-2 sequentially exactly $\frac{\|L\|}{2}$ times.

Parallel Bubblesort Example

1 3 8 5_1 6 7 2 5_2

Parallel Bubblesort Example

	1	3	8	5_1	6	7	2	5_2
Even Pass	1	3	5_1	8	6	7	2	5_2

Parallel Bubblesort Example

	1	3	8	5_1	6	7	2	5_2
Even Pass	1	3	5_1	8	6	7	2	5_2
Odd Pass	1	3	5_1	6	8	2	7	5_2

Parallel Bubblesort Example

	1	3	8	5_1	6	7	2	5_2
	1	3	5_1	8	6	7	2	5_2
	1	3	5_1	6	8	2	7	5_2
Even Pass	1	3	5_1	6	2	8	5_2	7

Parallel Bubblesort Example

	1	3	8	5_1	6	7	2	5_2
	1	3	5_1	8	6	7	2	5_2
	1	3	5_1	6	8	2	7	5_2
Even Pass	1	3	5_1	6	2	8	5_2	7
Odd Pass	1	3	5_1	2	6	5_2	8	7

Parallel Bubblesort Example

	1	3	8	5_1	6	7	2	5_2
	1	3	5_1	8	6	7	2	5_2
	1	3	5_1	6	8	2	7	5_2
	1	3	5_1	6	2	8	5_2	7
	1	3	5_1	2	6	5_2	8	7
Even Pass	1	3	2	5_1	5_2	6	7	8

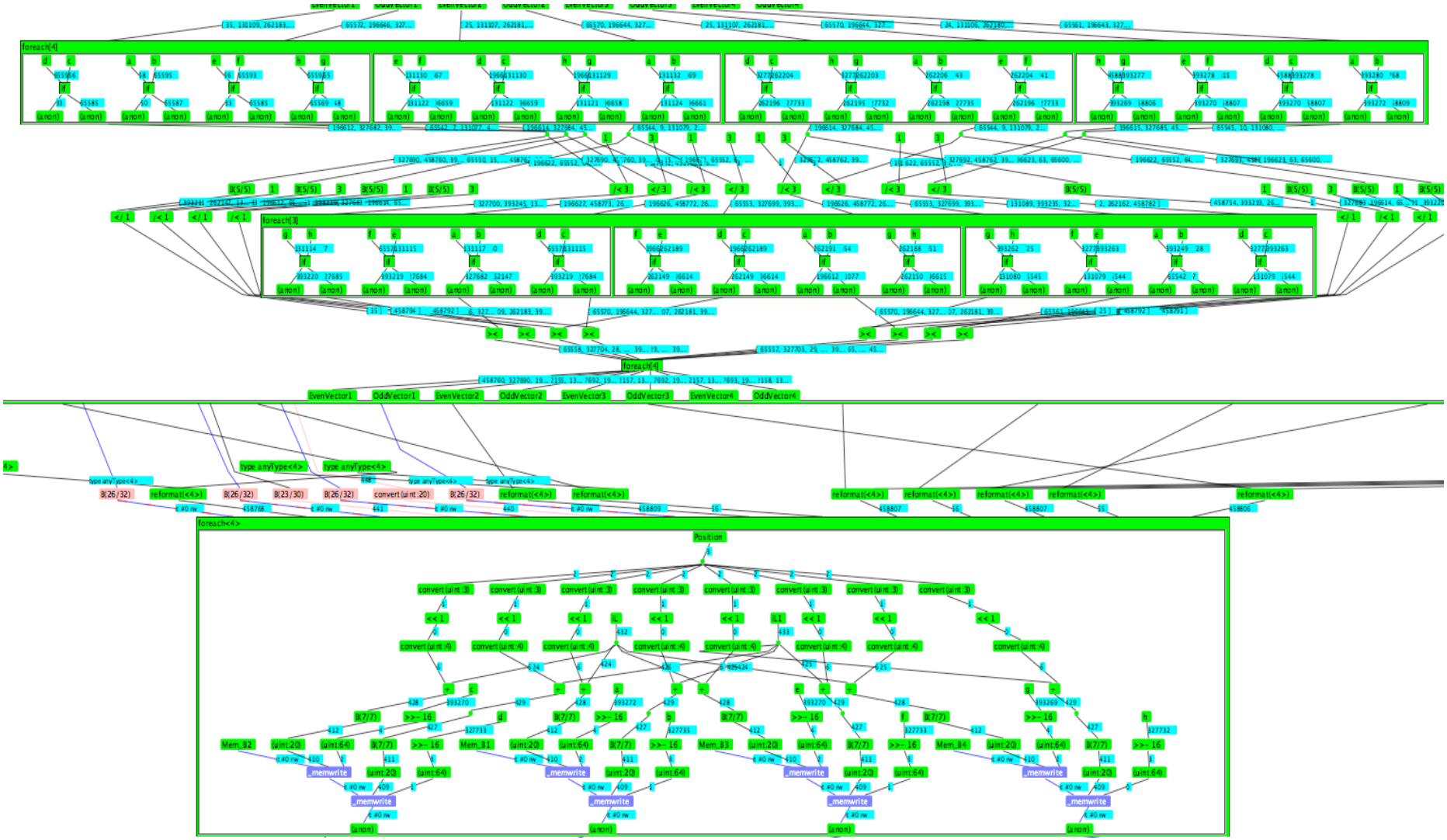
Parallel Bubblesort Example

	1	3	8	5_1	6	7	2	5_2
	1	3	5_1	8	6	7	2	5_2
	1	3	5_1	6	8	2	7	5_2
	1	3	5_1	6	2	8	5_2	7
	1	3	5_1	2	6	5_2	8	7
Even Pass	1	3	2	5_1	5_2	6	7	8
Odd Pass	1	2	3	5_1	5_2	6	7	8

Parallel Bubblesort in Mitrion-C

```
ELEMENT_TYPE[4] EvenVector = reformat( e, [4] );
ELEMENT_TYPE[4] OddVector = reformat( o, [4] );


(x, w) =  for( EvenOddPasses in  <1..4> ) {

   /** Pass: even part **/
   (EvensEP, OddsEP) =  foreach( a,b in EvenVector, OddVector ) {
       (p,q) = if( a > b ) (b,a) else (a,b);
    } (p,q);

   /** Pass: odd part **/
    (OddsOP,EvensOP) = foreach( a,b in (EvensEP /< 1),(OddsEP </ 3)) {
        (p,q) = if ( b > a ) (a,b) else (b,a);
    } (p,q);


    /* Add first and last elements back **/
    (EvenVector, OddVector) = foreach( a,b in
              ( EvensEP </ 1 ) >< EvensOP, OddsOP >< (OddsEP /< 3 ) ) {
    } (a,b);

} (EvenVector, OddVector);
```

Simulation of Parallel Bubblesort



Performance of Parallel Bubblesort

Length of List	Number of Lists	Speedup vs. qsort()
3	699048	12.0
4	524288	13.2
5	419428	16.4
6	349524	17.1
7	299592	18.9
8	262144	19.8
9	233016	21.5
10	209712	21.2
11	190648	22.6
12	174760	22.8
13	161316	23.9
14	149796	24.2
15	139808	24.9
16	131072	24.6

Performance of Parallel Bubblesort

Length of List	Number of Lists	Speedup vs. qsort()
14	149796	24.2
15	139808	24.9
16	131072	24.6
17	123360	25.4
18	116508	26.2
20	104856	25.9
24	87380	27.3
26	80656	28.4
32	65536	29.1
34	61680	29.8
36	58252	30.2
38	55188	30.3
39	53772	30.3
40	52428	30.0

Counting Sort Algorithm

Input: **L** a list
Output: **J** indices for sorted list

1. In parallel for each position in the list **L**, sequentially count the total number of elements less than the element appearing at this position plus the number of identical elements appearing before this position in **L**.
2. Return the sums.

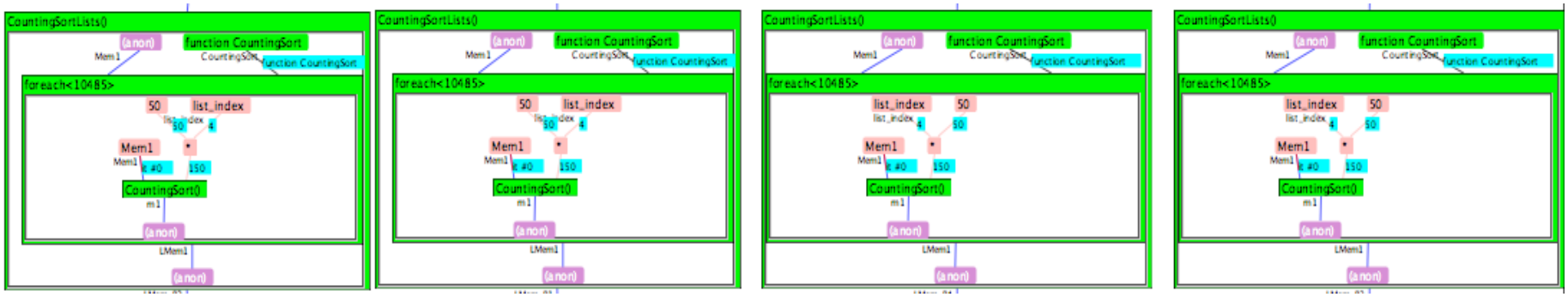
Example: Counting Sort

	1	3	8	5	6	7	2	5	
1									0
3	1						1		2
8	1	1		1	1	1	1	1	7
5	1	1					1		3
6	1	1		1			1	1	5
7	1	1		1	1		1	1	6
2	1								1
5	1	1		1			1		4

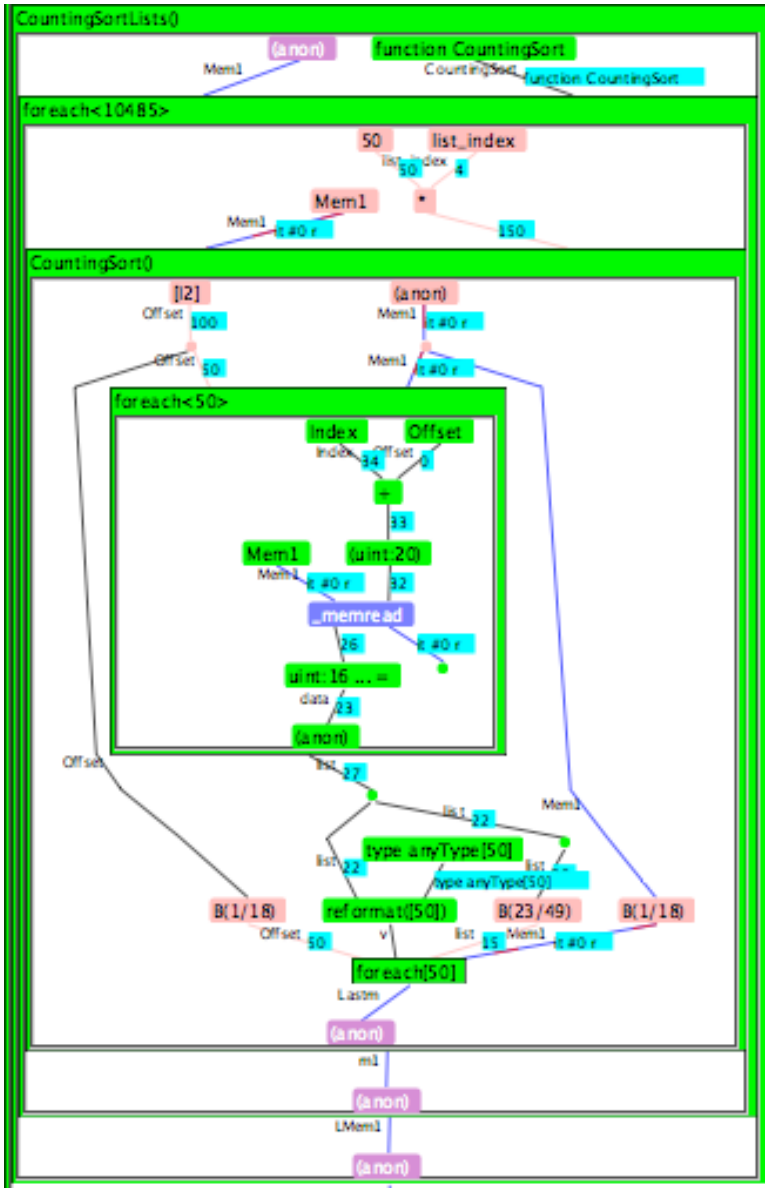
Counting Sort in Mitrion-C

```
PSCountingsort( Mem1, Offset )
{
  // Read list
  list = foreach( Index in <0.. L_1 > ) {
     SHORT_TYPE data = _memread( Mem1, Offset + Index );
  } data;

  v = reformat( list, [ L ] );

  // Perform counting sort:
  Lastm = foreach( e in v by INDEX_TYPE Pos_e ) {
    INDEX_TYPE sum = 0;
    Sum = for( Pos_a, a in <0.. L_1>, list ) {
      sum = if( a > e ) {
            } sum else {
               le = if ( (a==e) && (Pos_a >= Pos_e) ) {
                    } sum else {
                         s =sum + 1;
                    } s;
            } le;
      } sum;
    m  = _memwrite( Mem1, Offset + Sum, Pos_e );
  } m;
} Lastm;
```

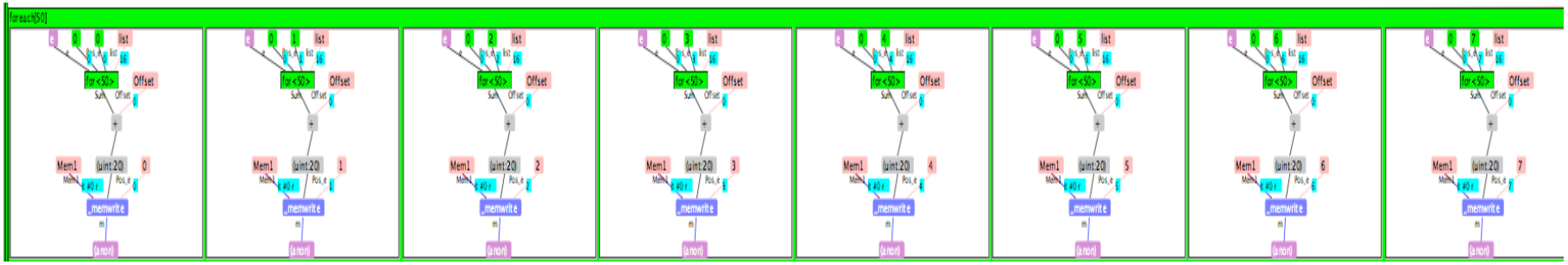
Simulation of Counting Sort



Simulation of Counting Sort



Simulation of Counting Sort



Performance of Counting Sort

Length of List	Number of Lists	Speedup vs. qsort()
3	699048	12.0
4	524288	13.2
5	419428	16.3
6	349524	17.0
7	299592	18.8
8	262144	19.9
9	233016	21.5
10	209712	21.3
11	190648	22.6
12	174760	22.9
13	161316	12.4
14	149796	12.6
16	131072	24.8
20	104856	26.0

Performance of Counting Sort

Length of List	Number of Lists	Speedup vs. qsort()
14	149796	12.6
16	131072	24.8
20	104856	26.0
24	87380	27.2
32	65536	29.1
36	58252	30.3
40	52428	30.0
46	45588	30.7
47	44620	31.4
48	43688	31.3
49	42796	31.7
50	41940	32.0

Alternating Mergesort Algorithm

Input: **L** a list
Output: **J** indices for sorted list
Requirement: $\|L\| = 2^k \cdot m$, for some k and m.

1. Merge lists of length m into lists of length $2m$ in alternating non-decreasing and nonincreasing orders. When comparing elements from a left and right list, choose the smaller element and in the case of equality, choose the element from the left list. If the right index has crossed into the left list, choose the leftmost element, which is the only choice by the invariant. If the left index has crossed into the right list, then choose the rightmost element by the invariant.
2. Double m and repeat step 2 as long as $2m \leq \|L\|$.

Alternating Mergesort Example

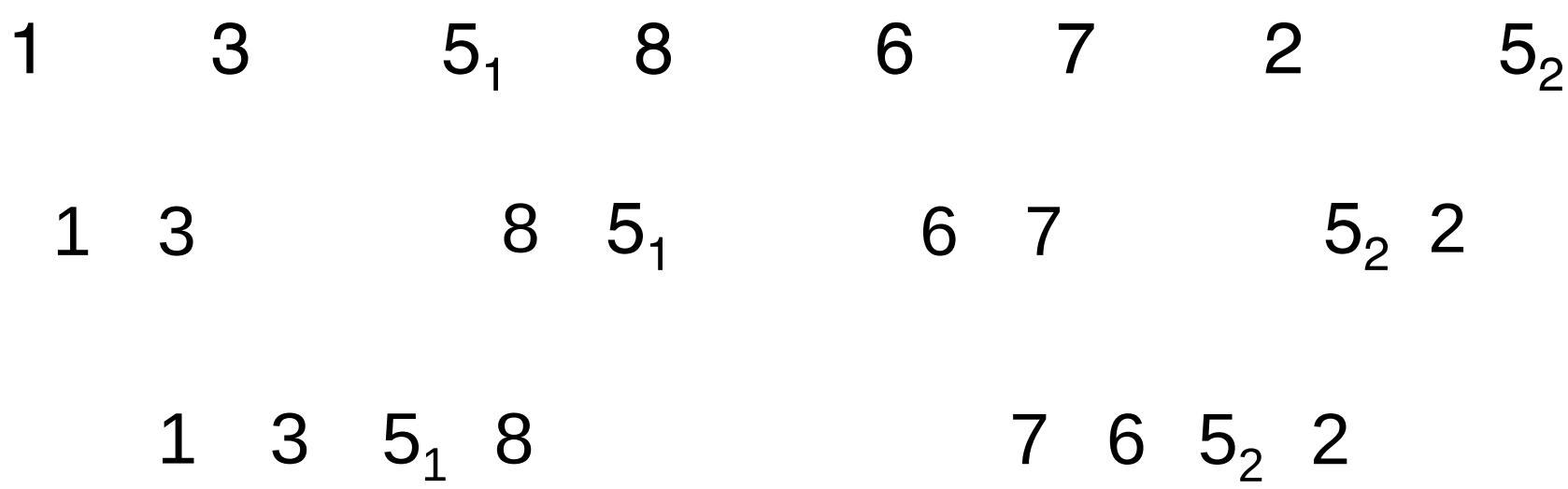
1 3 5_1 8 6 7 2 5_2

Alternating Mergesort Example

1 3 5_1 8 6 7 2 5_2

1 3 8 5_1 6 7 5_2 2

Alternating Mergesort Example



Alternating Mergesort Example

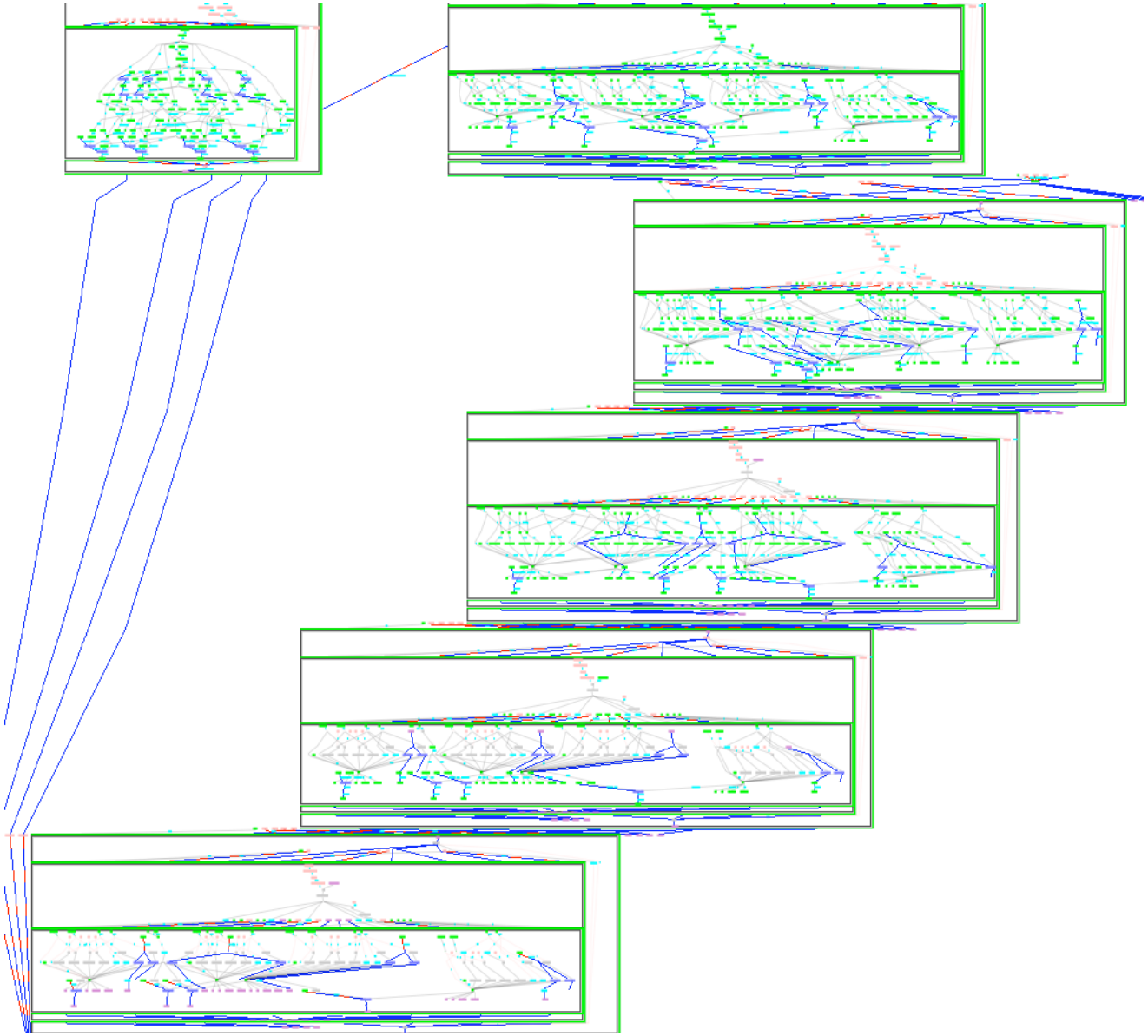
Body of Loop for Mergesort in Mitrion-C

```
x1 = _memread( itRD1, i1 );
(y1,itRD1) = _memread( itRD1, j1 );
OFFSET_TYPE jnext1 = j1 - 1;
OFFSET_TYPE inext1 = i1 + 1;

(i1,j1,xy1) = if( i1 >= m_2 ) {
              }  (i1, jnext1, y1)
              else {
                 (a,b,c) = if( j1 < m_2 ) {
                 } (inext1, j1, x1)
                 else {
                    (d,e,f) =
                       if( (DATA_TYPE)x1 > (DATA_TYPE)y1  ) {
                       } (i1, jnext1, y1)
                       else {
                       } (inext1, j1, x1);
                 } (d,e,f);
} (a,b,c);

Position = (ONE - ((ONE_BIT_TYPE)leaf_pair)*2)*p +
   ((ONE_BIT_TYPE)leaf_pair)*M - (ONE_BIT_TYPE)leaf_pair +1pM;
itWR1 = _memwrite( itWR1, Position, xy1 );
```

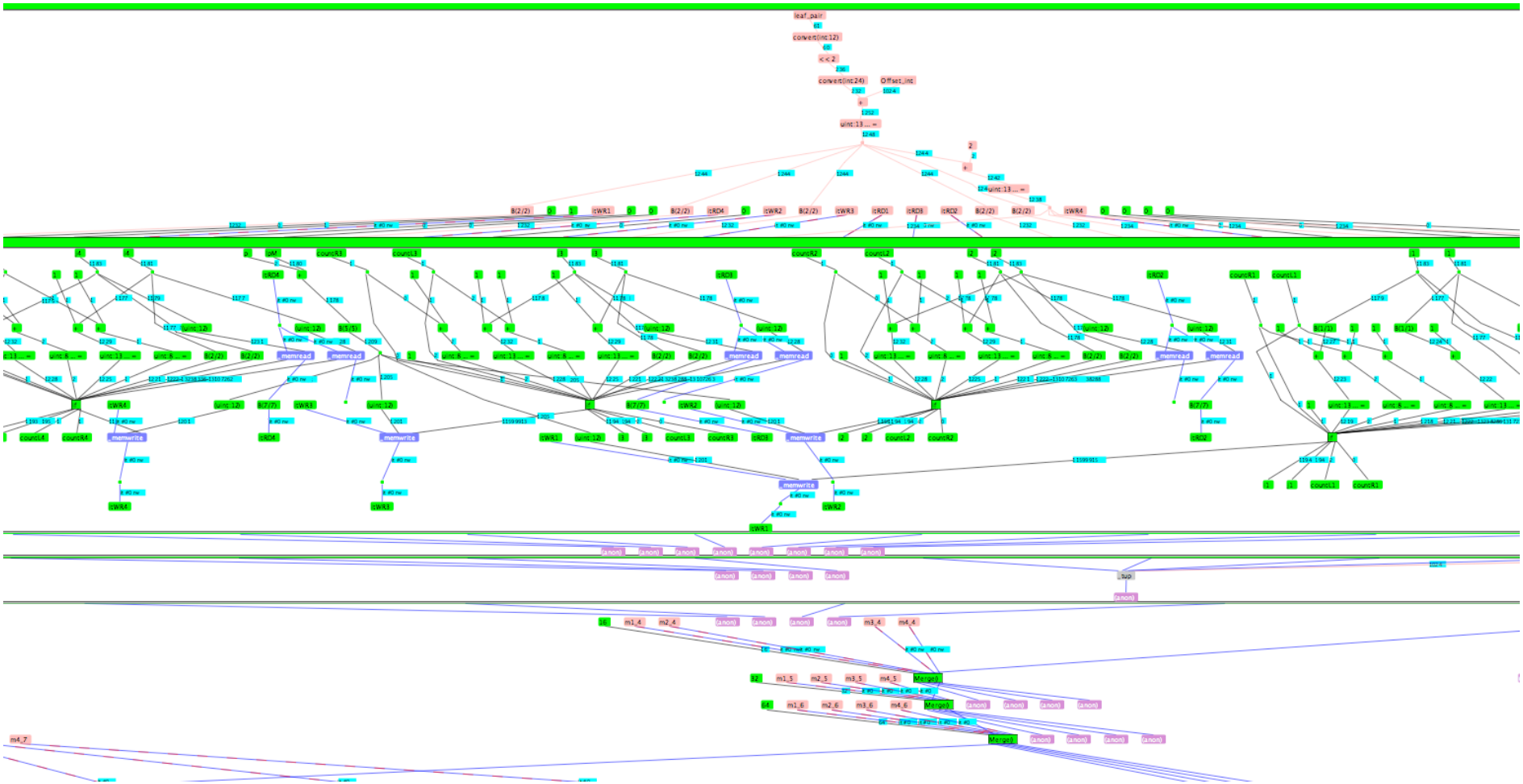
Body of Loop for Mergesort Using Counters

```
x1 = _memread( itRD1, i1 );
(y1,itRD1) = _memread( itRD1, j1 );
OFFSET_TYPE jnext1 = j1 + 1;
OFFSET_TYPE inext1 = i1 + 1;
INDEX_TYPE countLn1 = countL1 + 1;
INDEX_TYPE countRn1 = countR1 + 1;

(i1,j1,xy1,countL1,countR1) =
   if ((countL1 > M_21) || (countR1 > M_21)) {
      (q,r,s,t,u) =
         if( countL1 > M_21 ) {
         } (jnext1,jnext1,y1,countL1,countRn1)
         else {
         } (inext1,inext1,x1,countLn1,countR1);
   } (q,r,s,t,u)
   else {
      (q,r,s,t,u) =
        if( (DATA_TYPE)x1 > (DATA_TYPE)y1 ) {
            b=if(countRn1>M_21)i1 else jnext1;
        } (i1, b, y1, countL1, countRn1)
        else {
            a=if(countLn1>M_21) j1 else inext1;
        } (a, j1, x1, countLn1, countR1);
}(q,r,s,t,u);

Position = lpM+p;
itWR1 = _memwrite( itWR1, Position, xy1 );
```

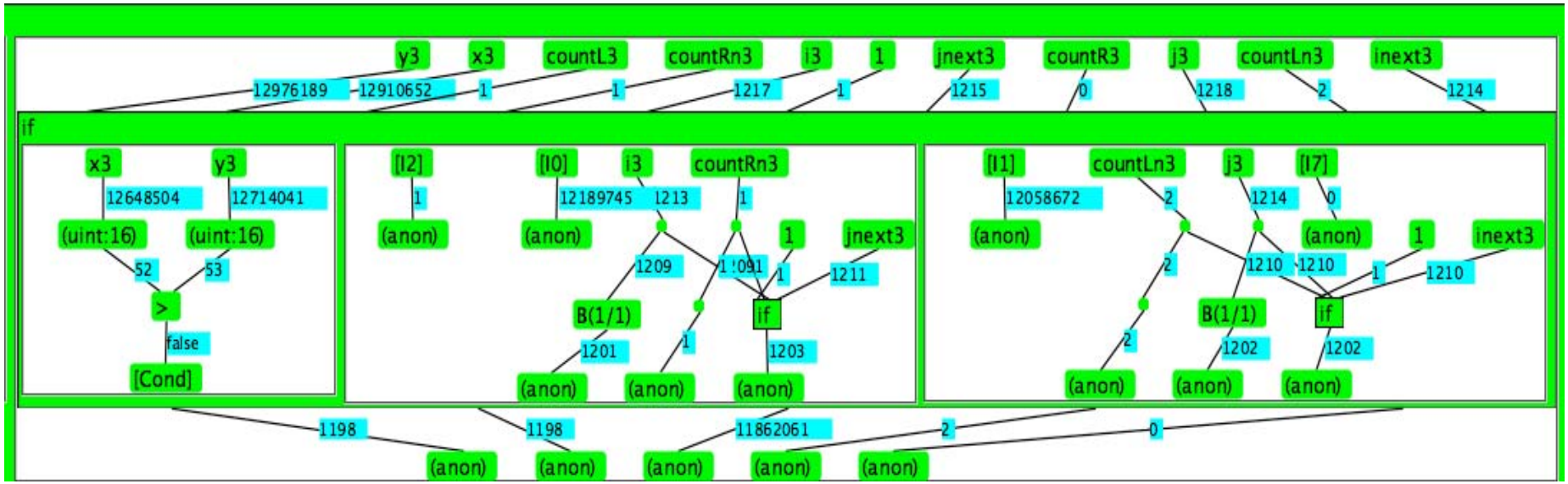
Simulation of Mergesort



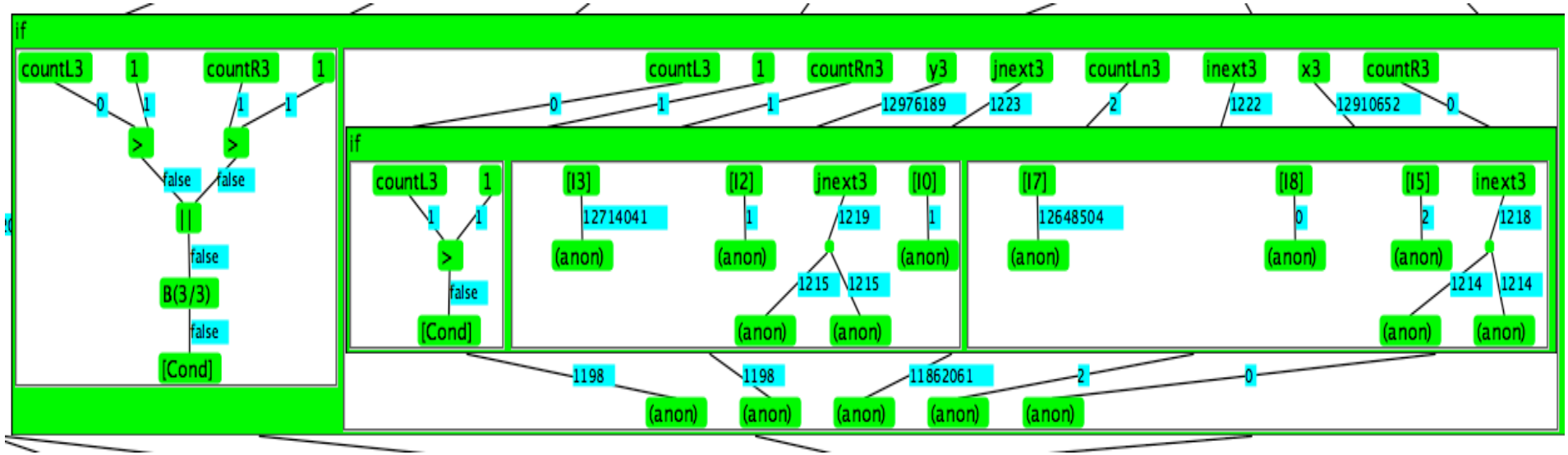
Simulation of Mergesort



Simulation of Mergesort



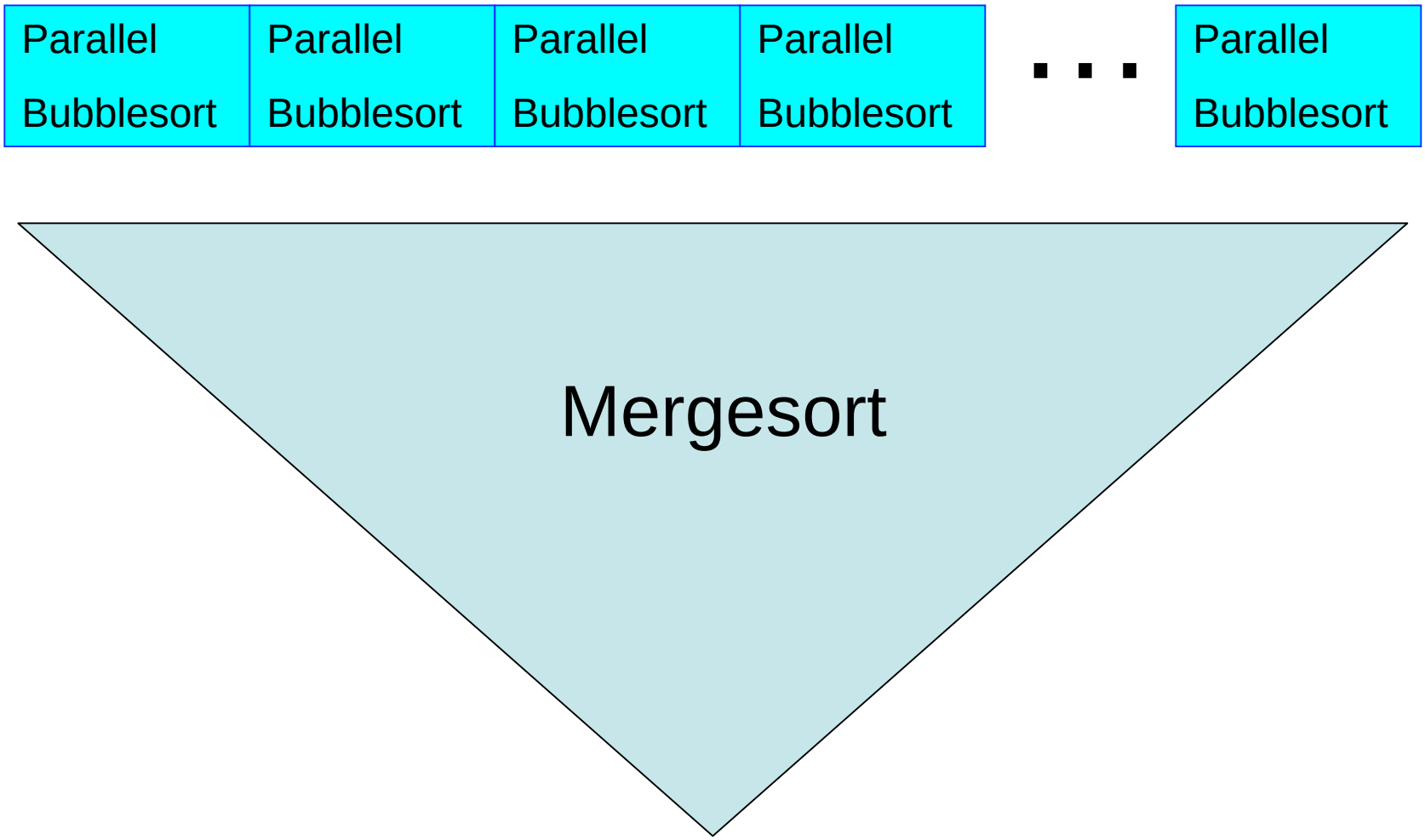
Simulation of Mergesort



Performance of Mergesort

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	16.2
128	16384	18.5
256	8192	20.0

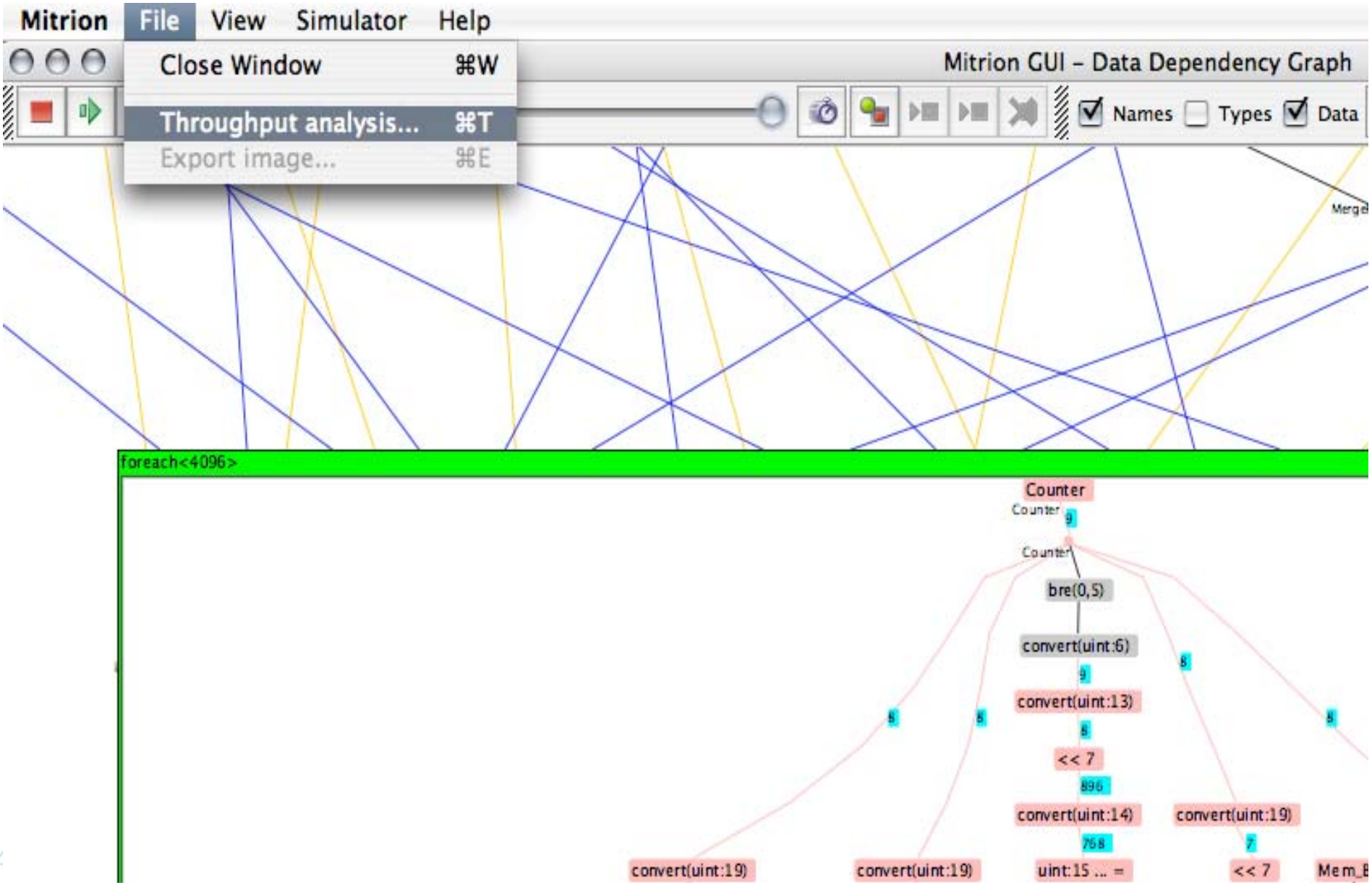
Parallel Bubblesort + Mergesort



Parallel Bubblesort + Mergesort = Good Fit?

- Try Throughput Analysis Window in Graphical Simulator
- Look at information on iteration structure, especially bottleneck values
  - relative speed of loops
  - low value $\Longrightarrow$ speed up

Simulation - Throughput Analysis



Simulation - Throughput Analysis

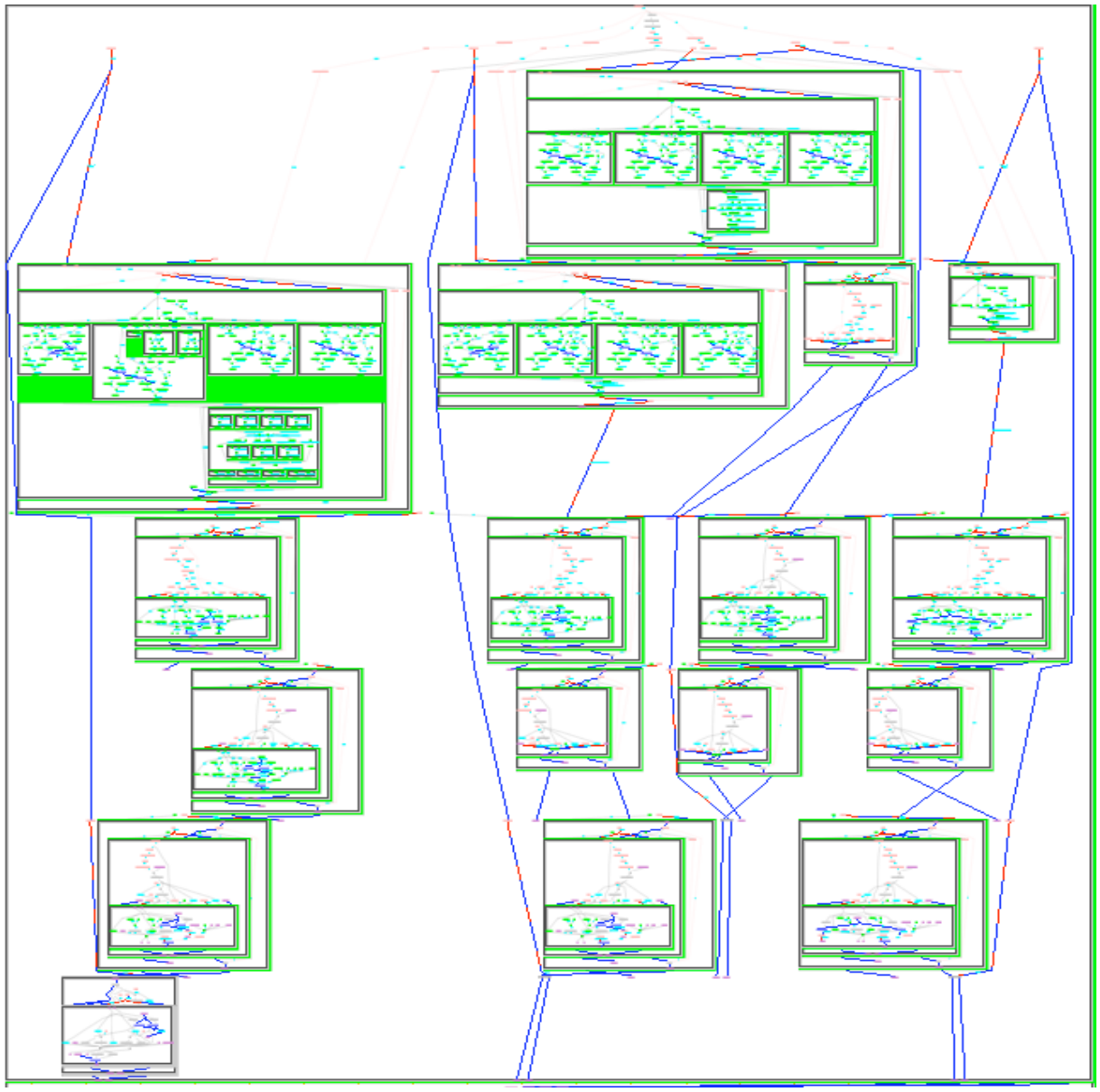
Mitrion File

Throughput Analysis

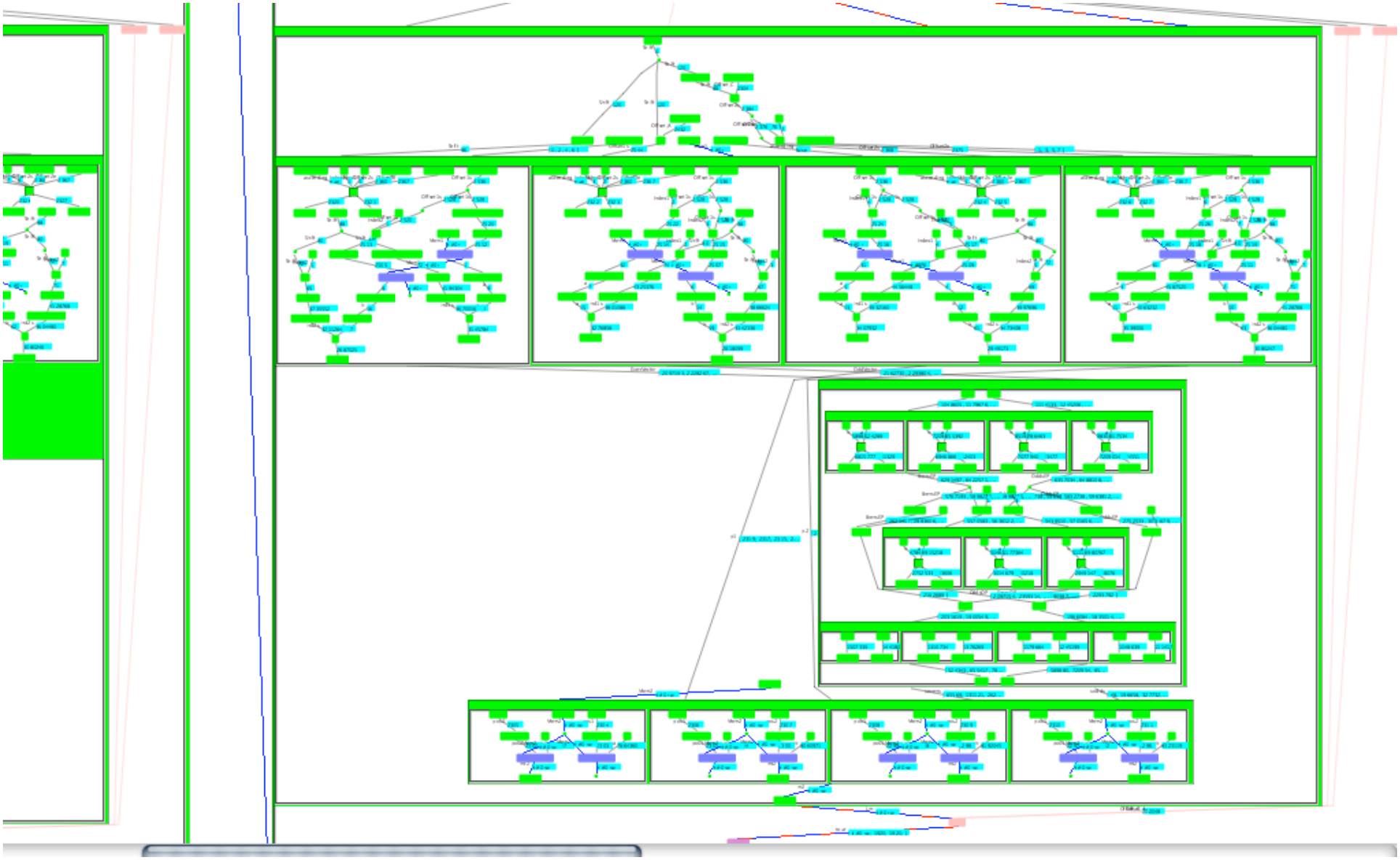
Iteration structure | Bottleneck Info

Loop Name	Iterations	Iteration Latency	Choke	Activation	Bottleneck
Throughput for main()	1		524288	524288	1
Main/main/foreach<4096>	4096		524288	128	1
ParallelBubblesort/foreach<16>	16		64	8	2
foreach[4]					
for<4>	4	4	4	4	2
foreach[4]					
Merge/foreach<8>	8		128	16	1
for<16>	16	9	16	9	1
Merge/foreach<4>	4		128	32	1
for<32>	32	9	32	9	1
Merge/foreach<2>	2		128	64	1
MergeFinal/for<128>	128	11	128	11	1
ParallelBubblesort/foreach<16>	16		64	8	2
Merge/foreach<8>	8		128	16	1
Merge/foreach<4>	4		128	32	1
Merge/foreach<2>	2		128	64	1
MergeFinal/for<128>	128	11	128	11	1
ParallelBubblesort/foreach<16>	16		64	8	2
Merge/foreach<8>	8		128	16	1
Merge/foreach<4>	4		128	32	1
Merge/foreach<2>	2		128	64	1
MergeFinal/for<128>	128	11	128	11	1
ParallelBubblesort/foreach<16>	16		64	8	2
Merge/foreach<8>	8		128	16	1
Merge/foreach<4>	4		128	32	1
Merge/foreach<2>	2		128	64	1
MergeFinal/for<128>	128	11	128	11	1

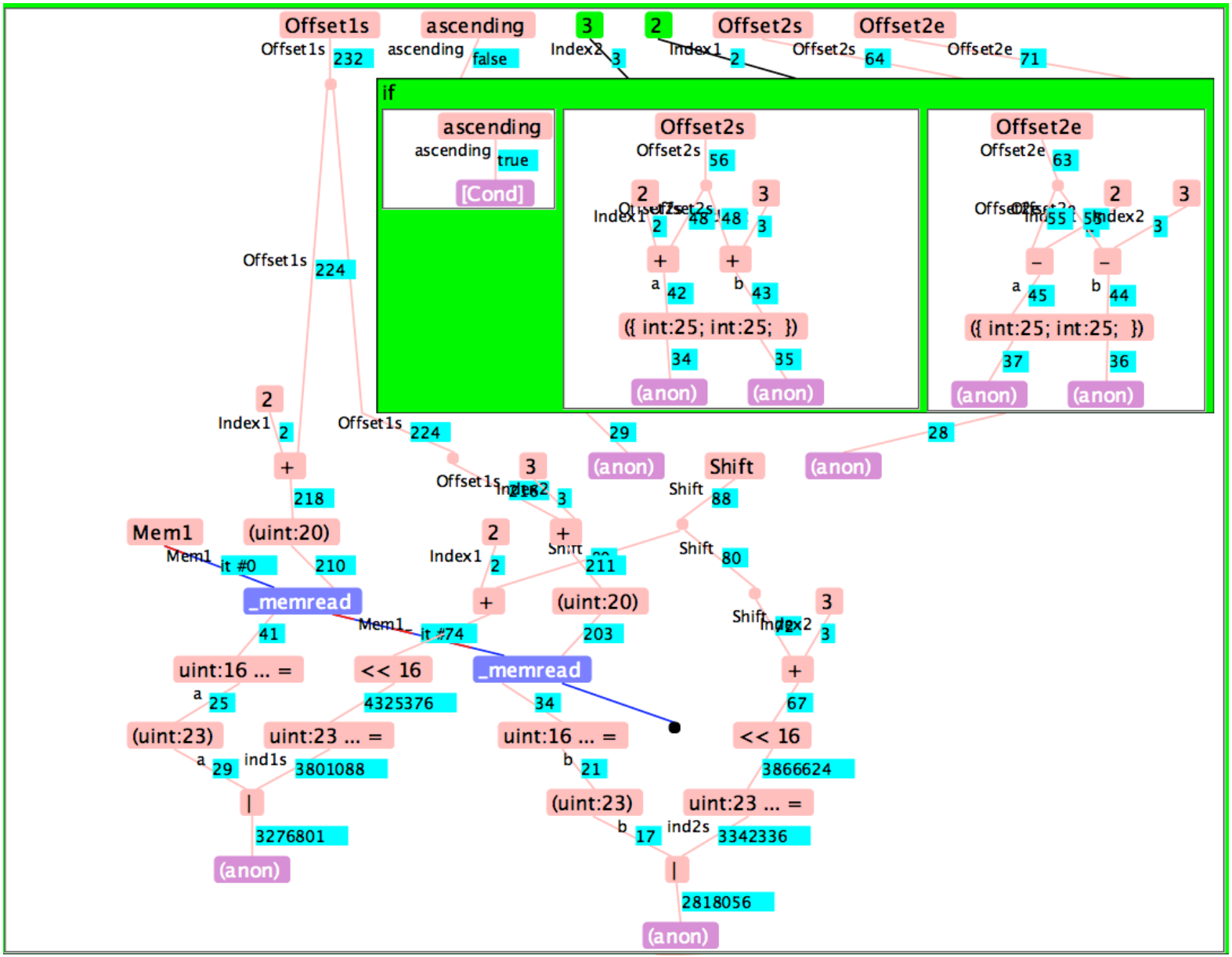
Simulation of Parallel Bubblesort and Mergesort



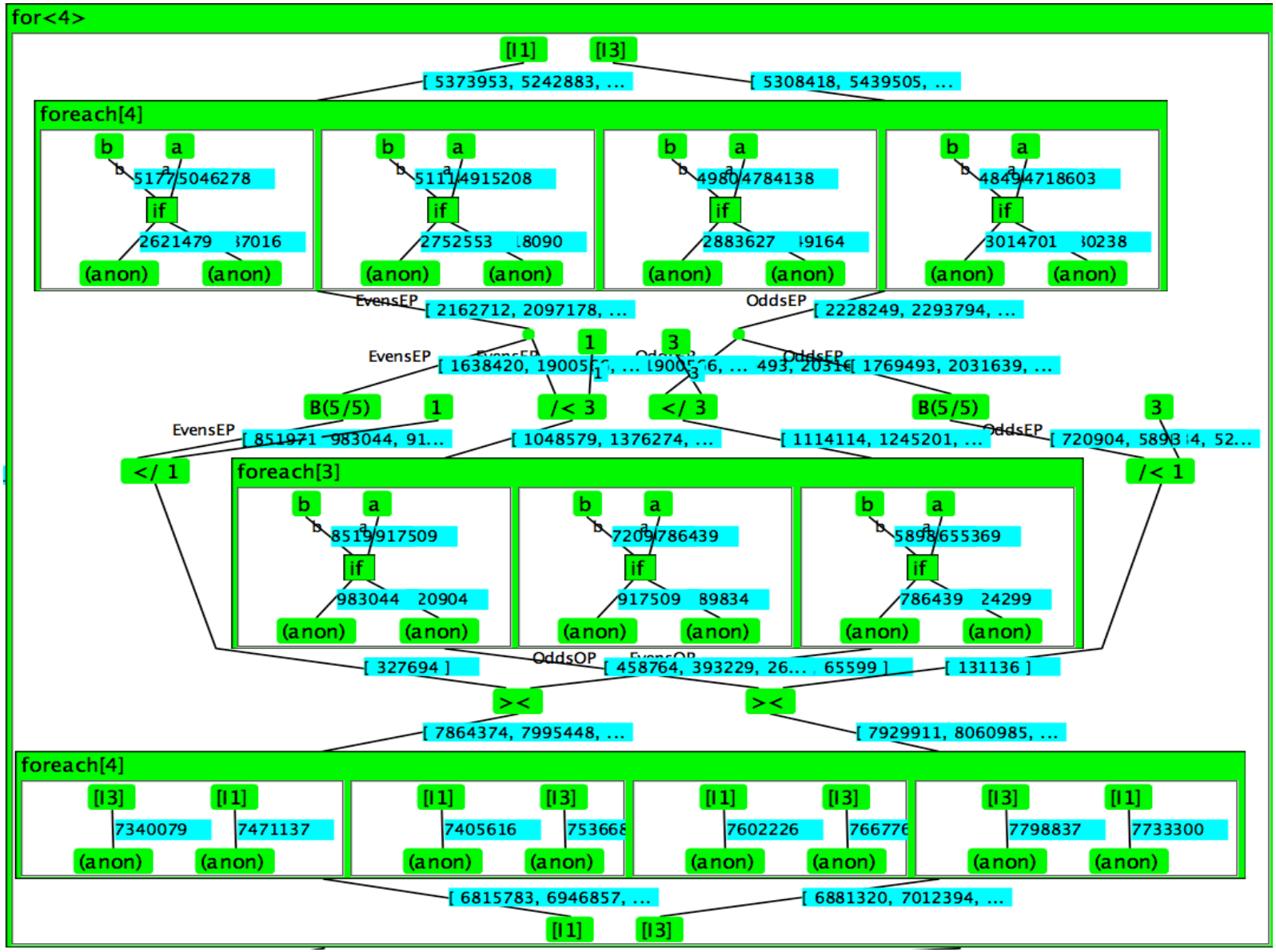
Simulation of Parallel Bubblesort and Mergesort



Simulation of Parallel Bubblesort and Mergesort



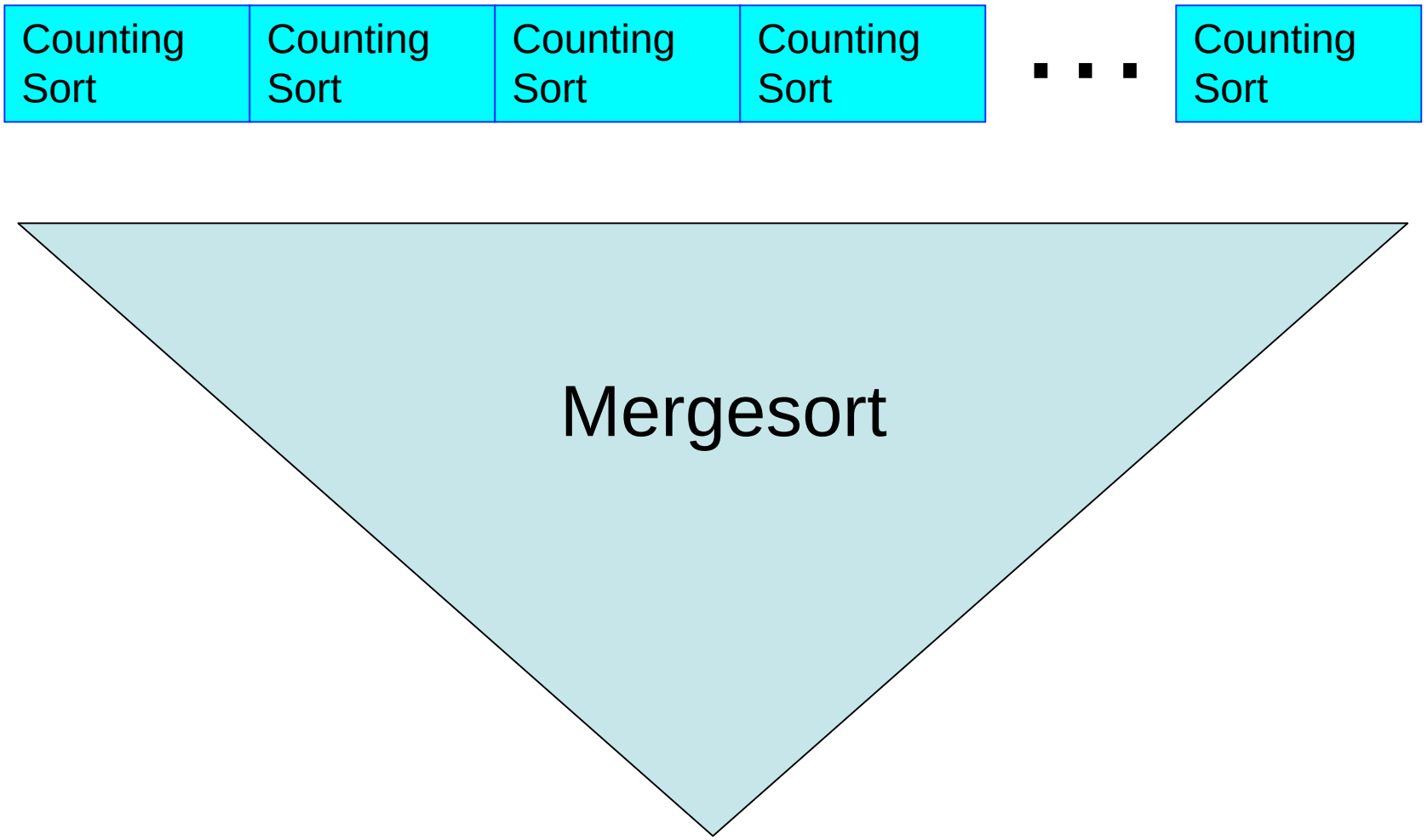
Simulation of Parallel Bubblesort and Mergesort



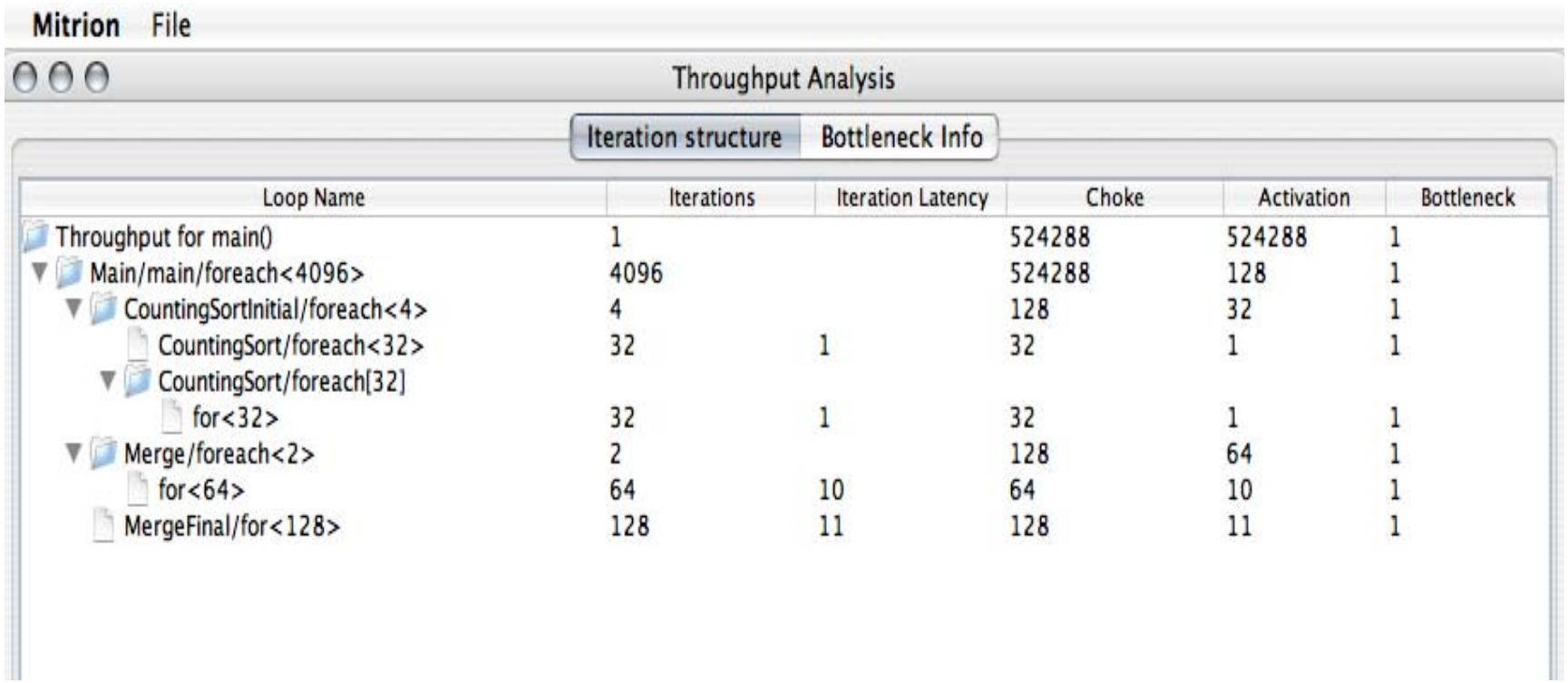
Performance of Parallel Bubblesort + Mergesort

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	9.9
128	16384	12.5

Counting Sort + Mergesort = Good Fit?

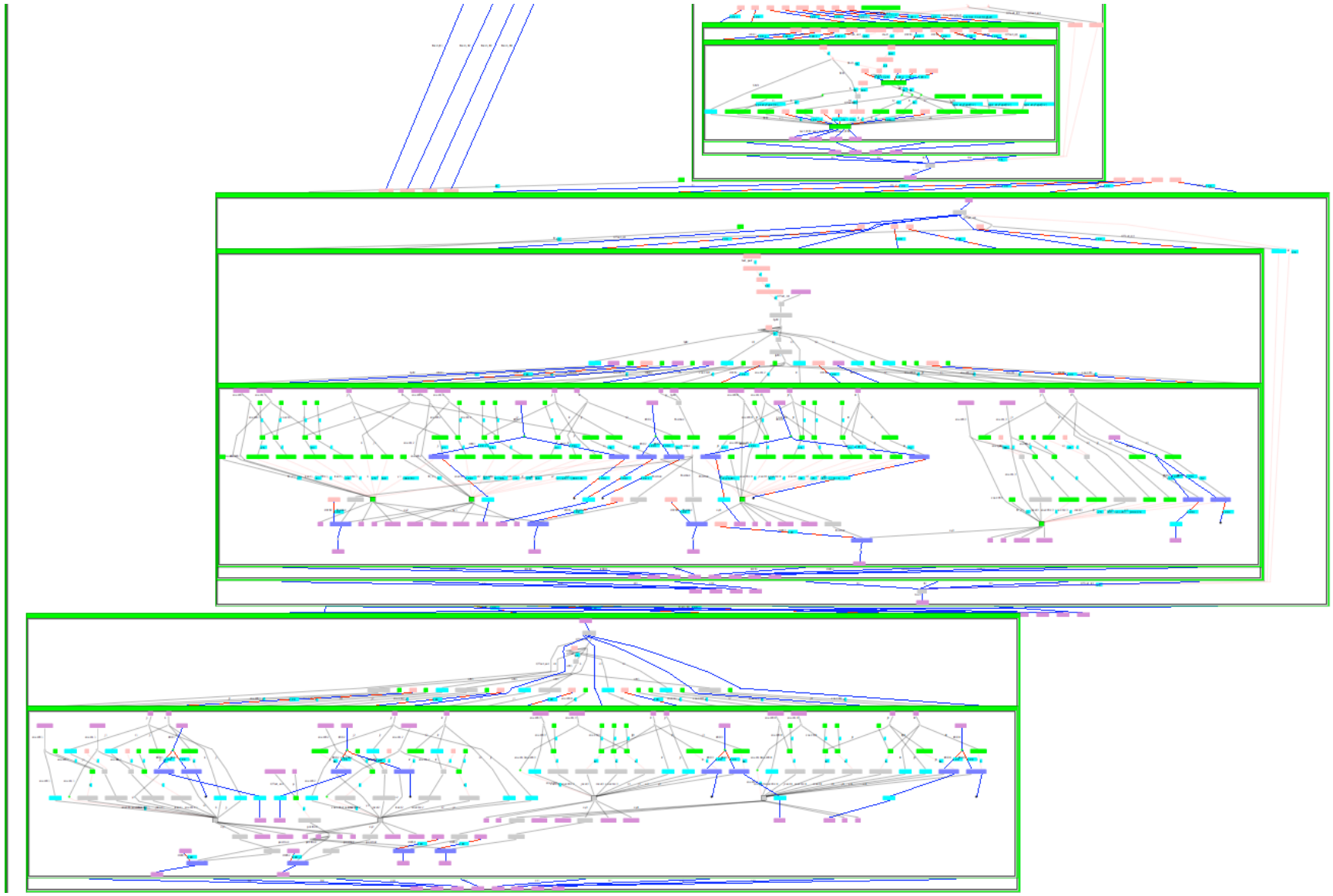


Simulation - Throughput Analysis

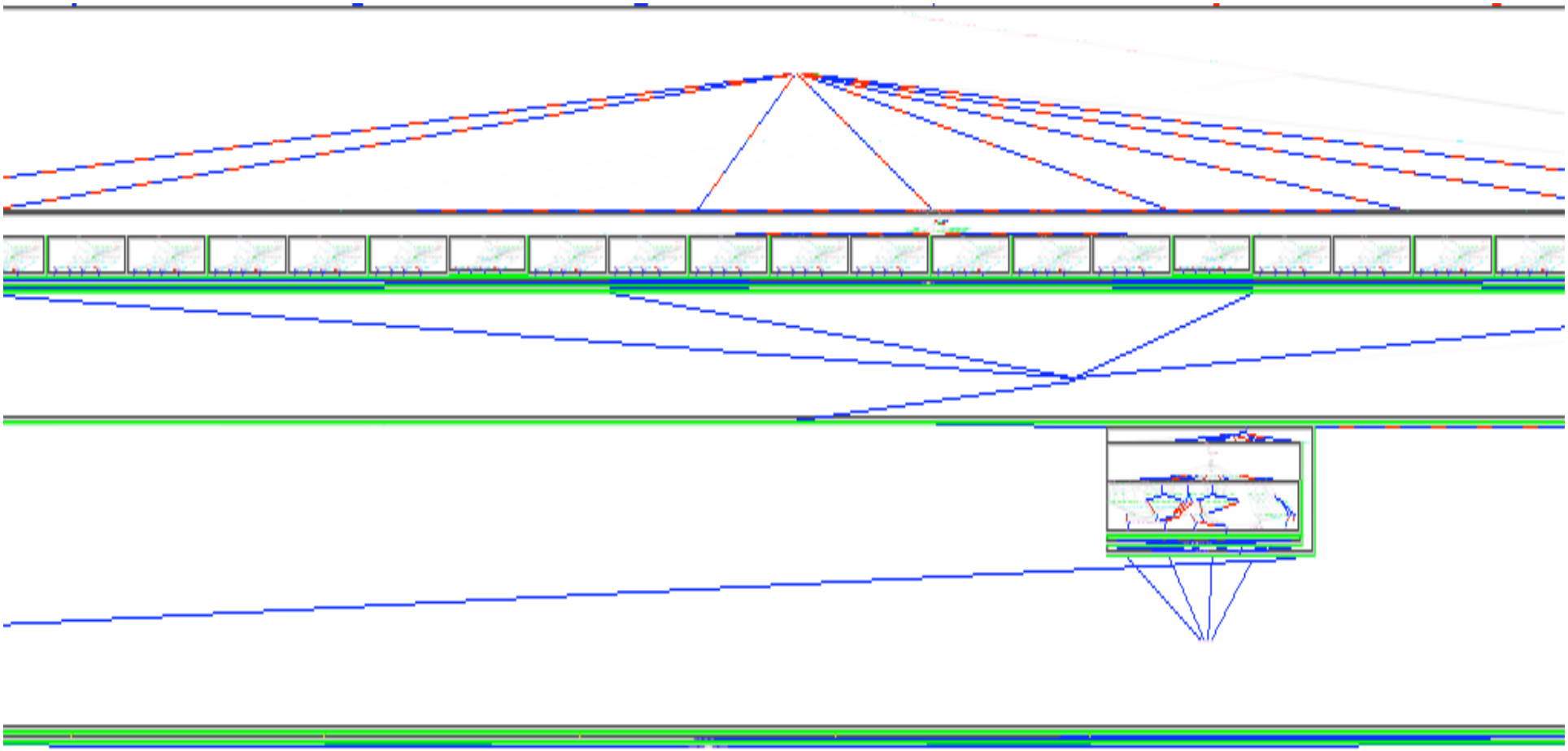


Loop Name	Iterations	Iteration Latency	Choke	Activation	Bottleneck
Throughput for main()	1		524288	524288	1
▼ Main/main/foreach<4096>	4096		524288	128	1
▼ CountingSortInitial/foreach<4>	4		128	32	1
CountingSort/foreach<32>	32	1	32	1	1
▼ CountingSort/foreach[32]					
for<32>	32	1	32	1	1
▼ Merge/foreach<2>	2		128	64	1
for<64>	64	10	64	10	1
MergeFinal/for<128>	128	11	128	11	1

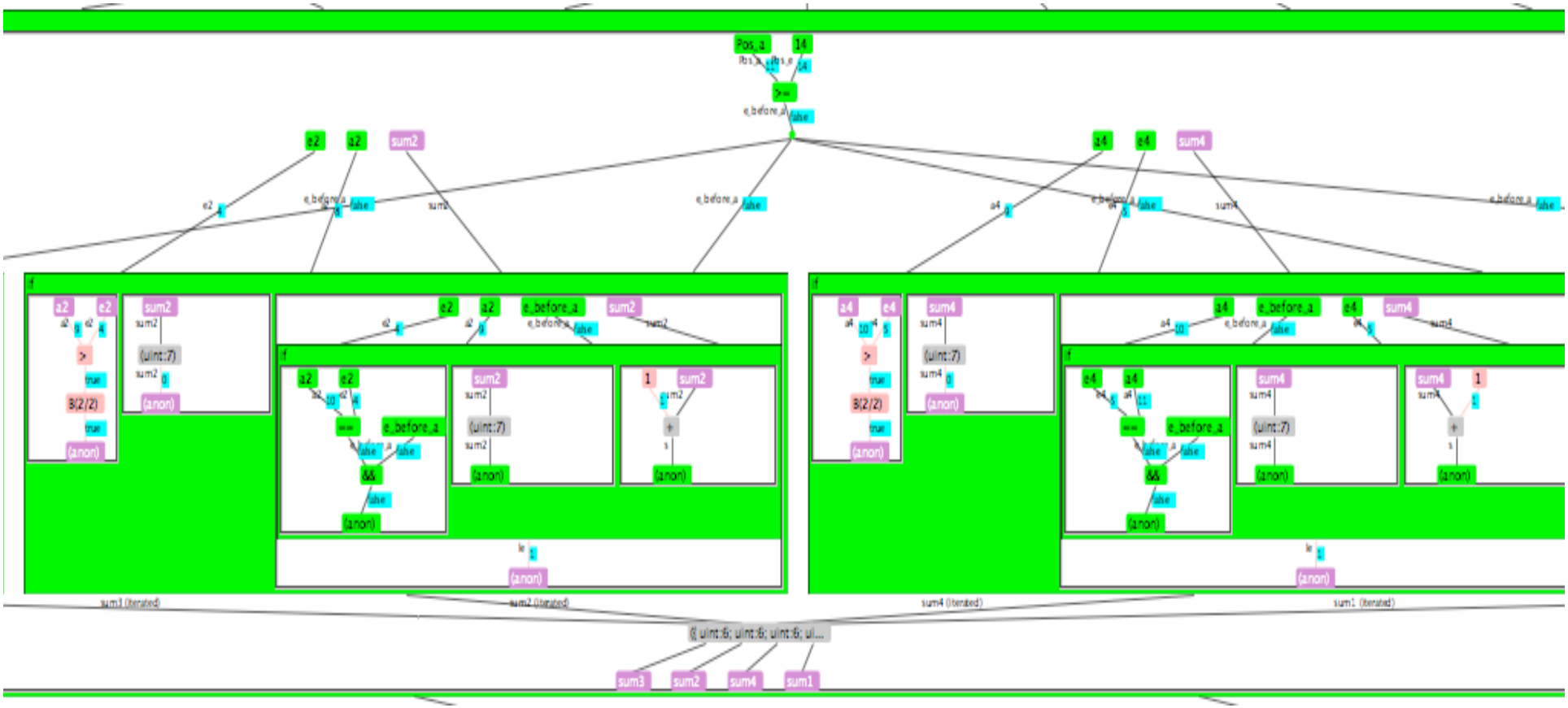
Simulation of Counting Sort and Mergesort



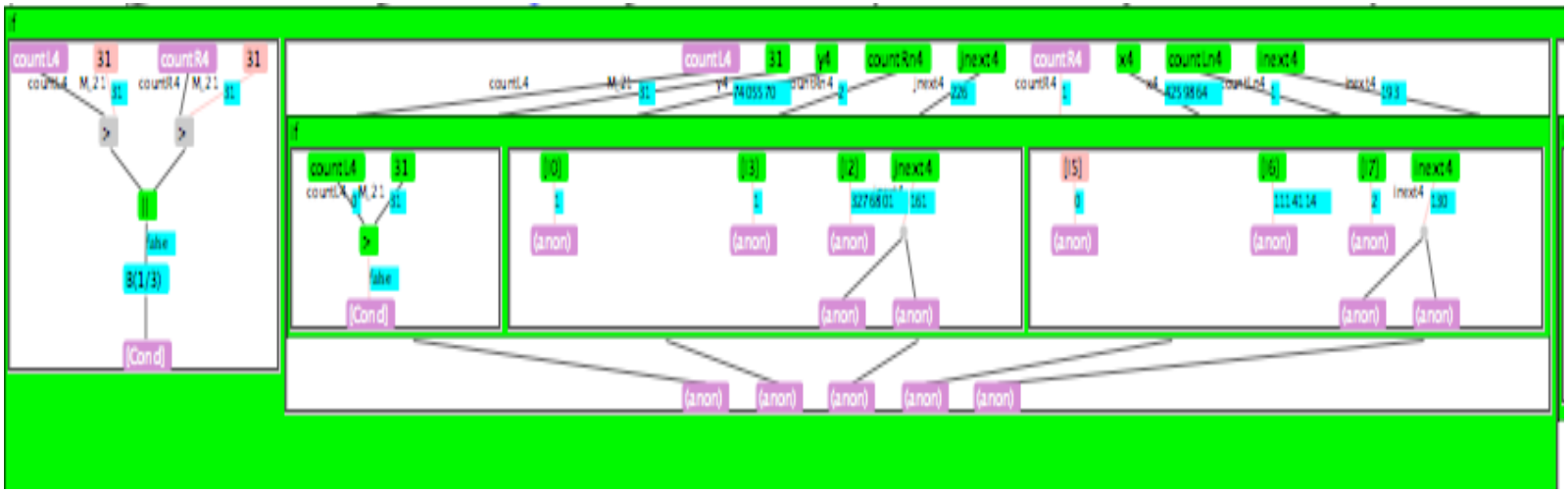
Simulation of Counting Sort and Mergesort

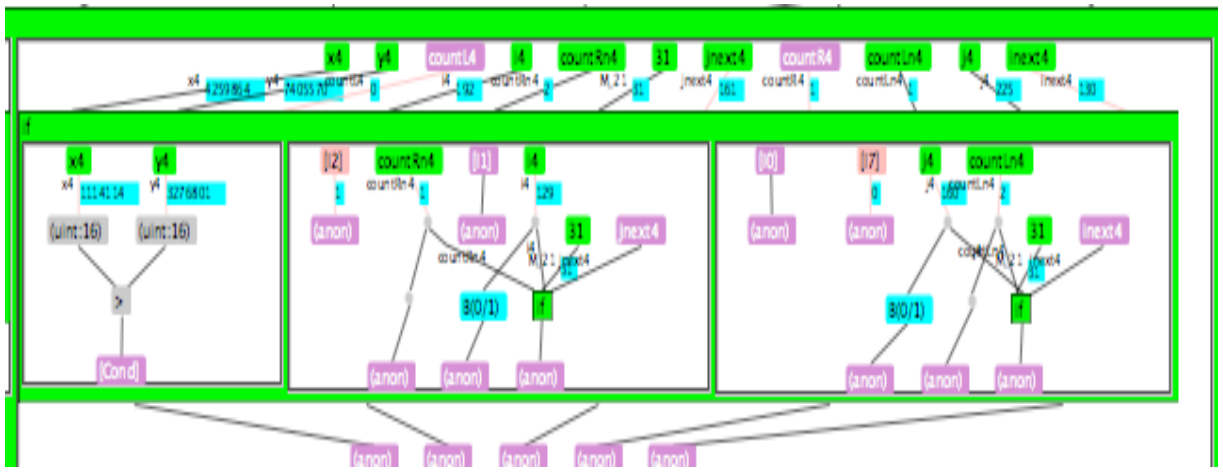


Simulation of Counting Sort and Mergesort



Simulation of Counting Sort and Mergesort





Performance of Counting Sort + Mergesort

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	11.0
128	16384	12.6

Summary of Speedups Observed

Length of List	Algorithm	Speedup vs. qsort()
3	Counting Sort	12.1
4	2-Way Selection Sort	13.8
5	Counting Sort	16.4
6	Parallel Bubblesort	17.1
7	Counting Sort	18.9
8	Counting Sort	19.9
9	Counting Sort	21.5
10	Counting Sort	21.3
11	Counting Sort	22.6
12	Counting Sort	22.9
13	Pipelined Counting Sort	24.0
14	Pipelined Counting Sort	24.4
16	Counting Sort	24.8
20	Counting Sort	26.0

Summary of Speedups Observed

Length of List	Algorithm	Speedup vs. qsort()
21	Counting Sort	26.8
24	Pipelined Counting Sort	27.3
32	Counting Sort	29.1
36	Counting Sort	30.3
39	Counting Sort	30.3
40	Counting Sort	30.1
46	Counting Sort	30.7
47	Counting Sort	31.4
48	Counting Sort	31.3
49	Counting Sort	31.7
50	Counting Sort	32.0
64	Mergesort	16.2
128	Mergesort	18.5
256	Mergesort	20.0