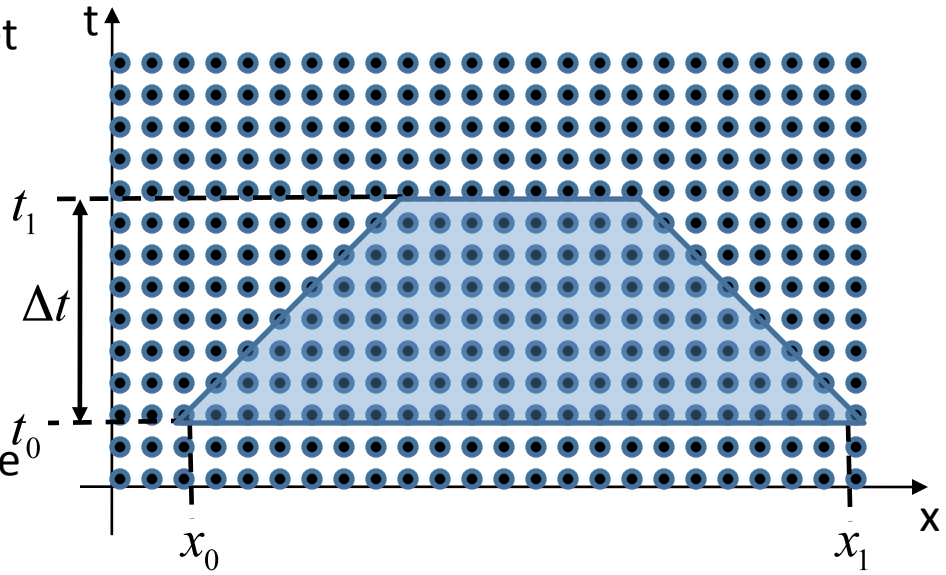
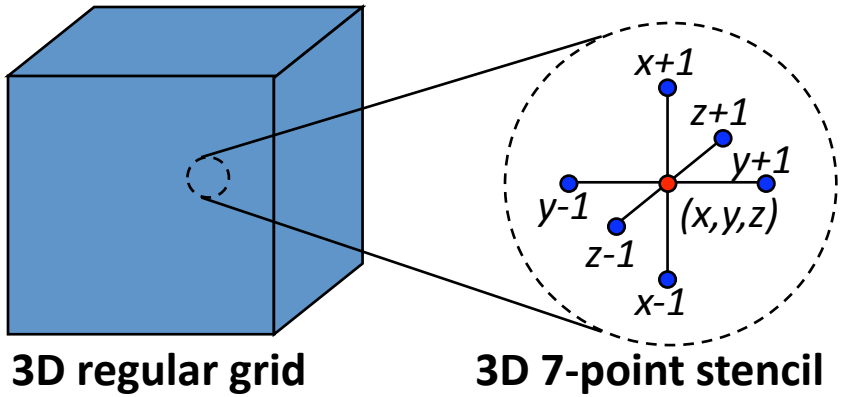
An Arbitrary N-dimensional Stencil Transformer in Cilk++

Yuan Tang, Steven Bartel, Dina Kachintseva

Background

- For a given point, a *stencil* is a fixed subset of nearest neighbors
- A *stencil code* updates every point in a regular/irregular grid by "applying a stencil"
- Formally, each point in an n-dimenstional spatial grid at time t is updated as a function of neighboring grid points at time t-1, …, t-k.
- Used in iterative PDE solvers like Jacobi, Multi-grid, and AMR
- Also used in areas like image processing and geometric modeling





3D regular grid **3D 7-point stencil**

Goals

- Based on Cilk++ technology, we will develop a stencil compiler that accomplishes the following:
  - Faster
  - Generalized to arbitrary N-dimensional space
  - Deals with boundary conditions
  - Irregular shaped boundary conditions

Summary

Optimizations

- Coarsening Strategy (16%)
- Cutting Heuristics (12%)
- Zero Padding (25%)
- SIMD + Loop Unrolling (10%)
- Optimize ‘N’ (42%)

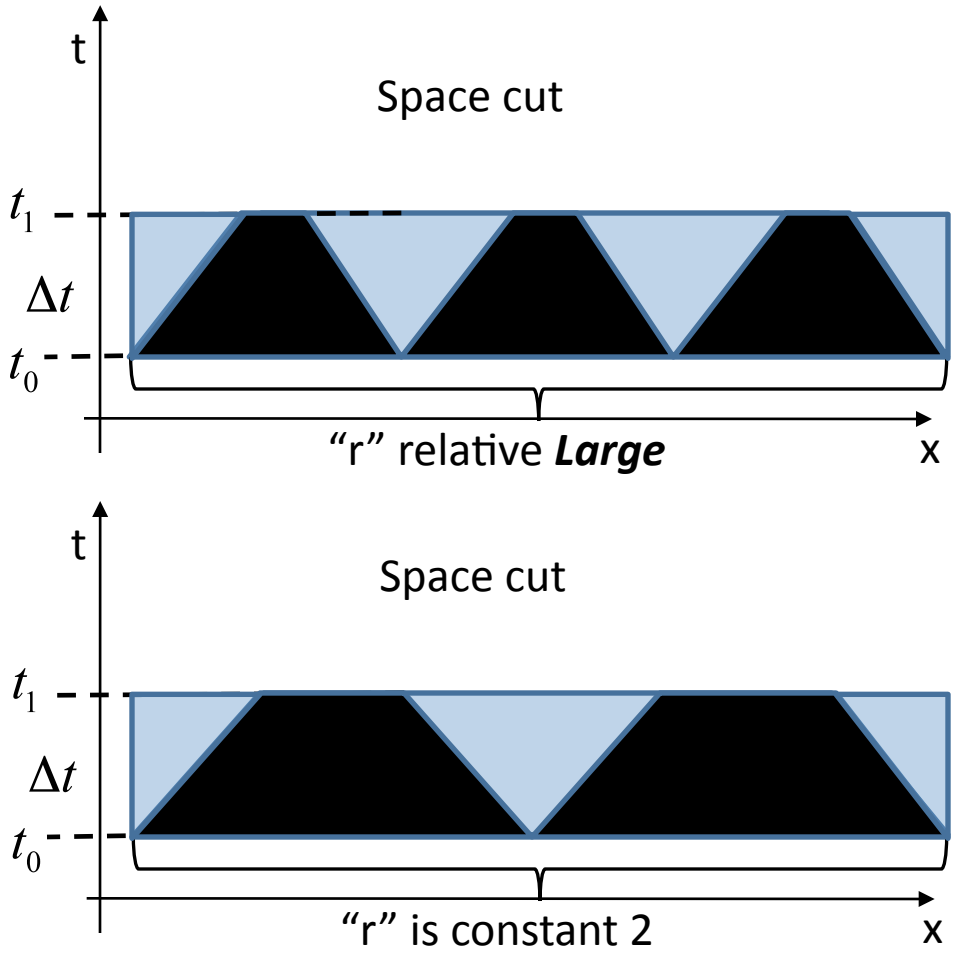
Key Features

- Arbitrary N-dimension
- Irregular Grids
- Periodic vs. Non-Periodic
- SIMD + Loop Unrolling
- Static Elaboration

2.6 times faster than cilk_for

Faster : Cutting Heuristics

- Cutting heuristics for cache-oblivious algorithm:
  - In Matteo Frigo's original paper, when it comes to a space cut, it cuts a space dimension into "r" black trapezoids and "r+1" of gray trapezoids. And "r" should be relative ***Large***.
  - We conducted each space cut into only Two (2) "black" trapezoids, and Three (3) "Gray" trapezoids.
    - Perform better on less-core machines than on more-core machines
    - Cache locality .vs. parallelism
    - Requires more theoretical analysis
  - ***Heuristics***: Cutting in spatial dimension should be proportional to the number of cores available? ---- processor aware?

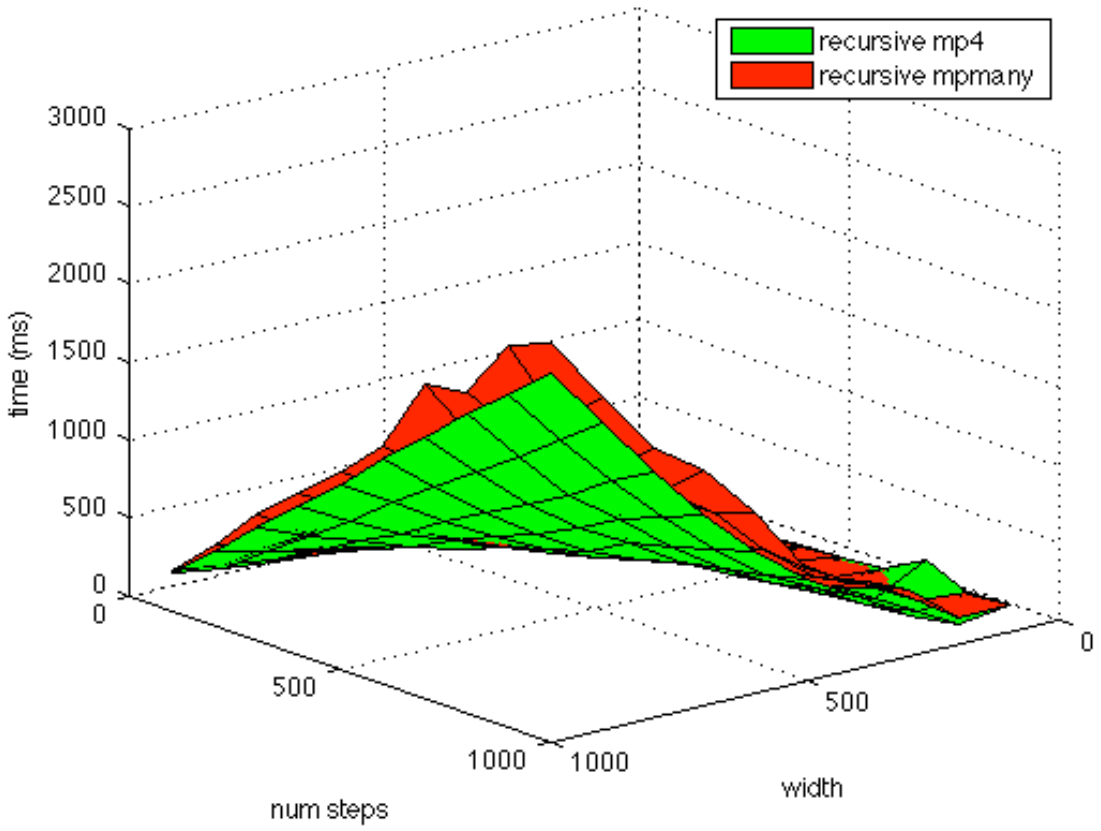


Faster : Cutting Heuristics

- Test number of trapezoids per space cut. NCuts = 2,4,8,16,as many as we can.
- Ran simulations on 100x100grid, 100 time steps to 1000x1000grid, 1000 time steps.
- 4, 8 performs the best (539ms, 552ms on average) compared to 2, 16, many (615ms, 622ms, 613ms on average).
- 12% speedup.

Faster : Cutting Heuristics

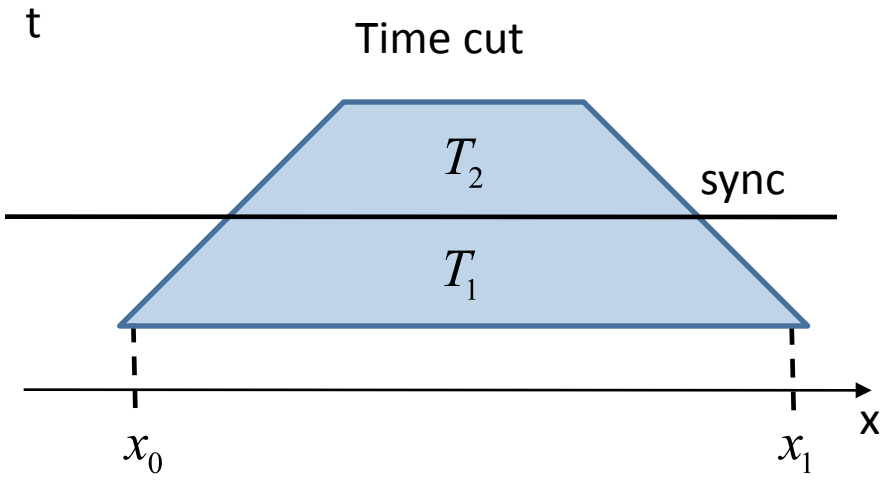
	Recursive mp4	Recursive mpmany
Range - 100x100 to 1000x1000	17.49 ms to 2258.0 ms	18.36 ms to 2547.8 ms
Average	539.12 ms	613.50 ms



Faster : Coarsening strategy

- Base case:
  - Step into base case computation Only when “lt=1”.
    - Implication: coarsen is conducted in time dimension Only.
    - Always cut into space before any attempt on time dimension: the space cube may be too small when cut into time ---- lose the benefits of coarsening in space .
    - The computation of base case can only be triggered by previous cut in time, which is severely serialized in execution.
    - Even worse, imagining an extreme case: ./heat 1 billion 1 (1 billion x 1 billion grid, and only 1 time step).
  - We step into base case depending on the size of all space and time dimensions. Moreover, prevent small cut into space dimensions.

```
Walk(…) {
If (lt <= T_STOP) {
  base_case(…);
} else if (lt > T_STOP) {
  if (x1 – x0 > 4 * slope_x * lt) {
    /* cut into X dimension */
    …..
  } else {
    /* cut into Time dimension */
    walk(t0, t0+lt/2, …);
    walk(t0+lt/2, t1, …);
  }}}
```



Faster : Coarsening strategy

Original

```
Walk(…) {
If (lt <= T_STOP) {
  base_case(…);
} else if (lt > T_STOP) {
  if (lx > 4 * slope_x * lt) {
     /* cut into X dimension */
     …..
  } else {
     /* cut into T dimension */
     walk(t0, t0+lt/2, …);
     walk(t0+lt/2, t1, …);
  }
}
}
```

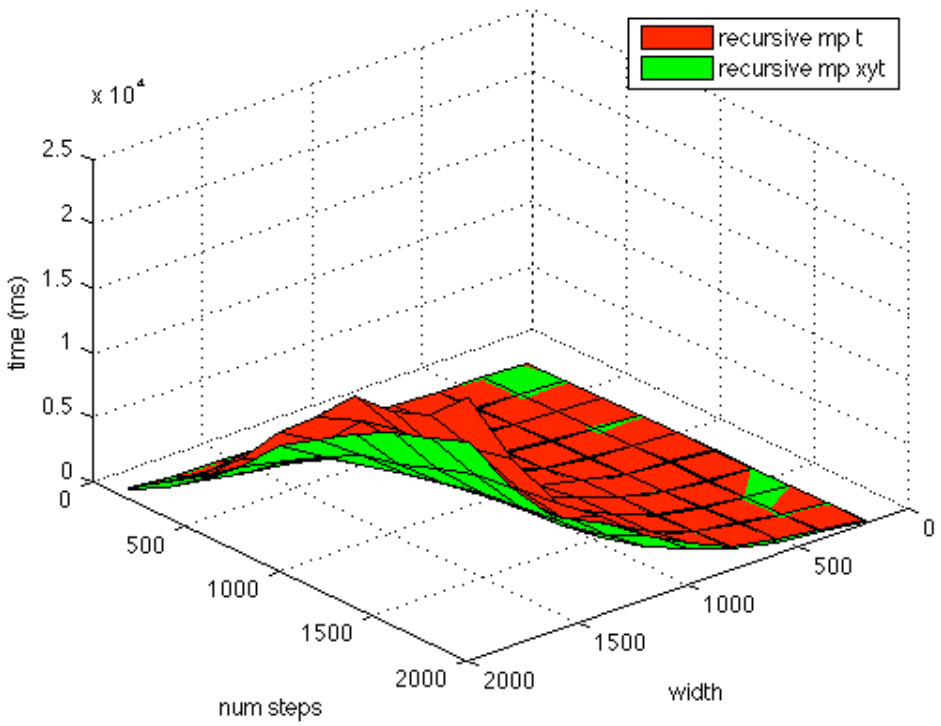
Refined

```
Walk(…) {
If (lt <= T_STOP && lx <= X_STOP && ly <= Y_STOP) {
  base_case(…);
} else {
  if (lx> 4 * slope_x * lt && lx > X_STOP) {
     /* cut into X dimension */
     cilk_spawn walk(black1);
     walk(black2);
     cilk_sync;
     cilk_spawn walk(gray1);
     cilk_spawn walk(gray2);
     walk(gray3);
     return;
  } else if (ly > 4 * slope_y * lt && ly > Y_STOP){
     /* cut into Y dimension */
     ……
  } else {
     /* cut into T dimension */
     walk(t0, t0+lt/2, …);
     walk(t0+lt/2, t1, …);
  }}}
```

Coarsen on t vs xyt

- Speed of stop on lt only vs. speed of stop on all space and time dimensions. As width and num steps increase, so do the gains from additional coarsening parameters.

	Recursive mp t	Recursive mp xyt
Range - 200x200 to 2000x2000	20ms to 20379ms	20ms to 16960ms
Average	3831 ms	3236 ms



SIMDizing the Base Kernel

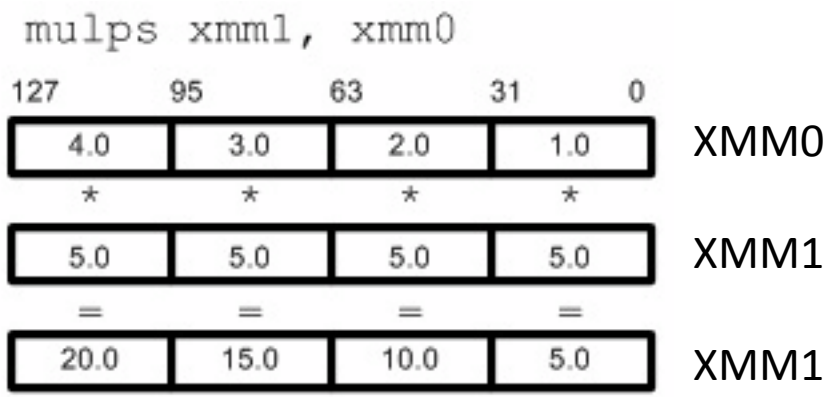
- Want to speedup the base_kernel calculations by:
  - Replacing C++ floating point calculations with corresponding SIMD instructions

Original 2D code:

```
U(Q, t+1,x,y) = Q->CX * (U(Q, t,x+1,y) - 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
    + Q->CY * (U(Q, t,x,y+1) - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
    + U(Q, t,x,y);
```

3 sets of floating point operations that can be done in parallel for the X and Y dimensions

SIMDizing the Base Kernel

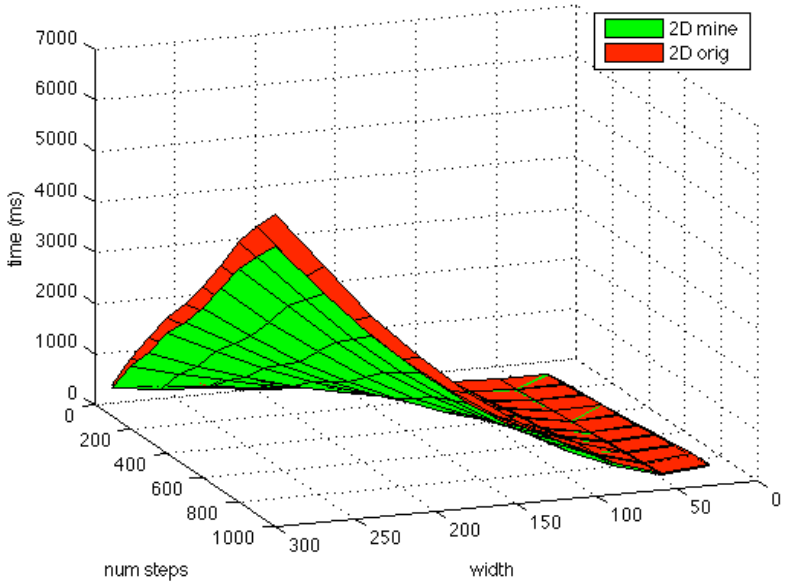



```
double vec1[2] __attribute__((aligned(16)));
double vec2[2] __attribute__((aligned(16)));
.....
vec1[0] = U(Q, t,x+1,y);
vec2[0] = U(Q, t,x-1,y);
.....
asm volatile(
        "movapd (%0), %%xmm0\n\t"
        "addpd (%1), %%xmm0\n\t"
        "subpd (%2), %%xmm0\n\t"
        "mulpd (%3), %%xmm0\n\t"
        "movapd %%xmm0, (%0)"
        ::"r"(vec1),  "r"(vec2), "r"(twovec), "r"(cvec)
      );
```

SIMDizing the Base Kernel

- Replacing the C double operations with SIMD instructions created a speedup of 10%

Loops	Original	SIMD
Range – 100x100 to 1000x1000	41 ms to 6159 ms	37 ms to 5521 ms
Average	1378 ms	1237 ms



Generalize to Arbitrary N-dimension

- 'N' is not known at program-time, but known at compile-time.
- Traversal of all elements in grid is awkward.
- Too many redundant copy of the index.
- Even with all optimization applied, it's still much slower than specific version
- Up to 8 dimensions, 10^8 grids can be computed : consuming $2 \times (size + 2 \times slope)^N \times 8$ bytes – only constraint is the amount of memory system can allocate to user applications.
- Needs optimizing 'N' – meta-programming (currently Python scripts)

```
Base_case_kernel(t0, t1, grid) {
   for (int t = t0; t < t1; t++) {
        while (!done) {
            kernel(t, l_index);
            done = update_index(l_index, l_head_index, l_tail_index);
        }
   }
}
Update_index(idx, const head_idx, const tail_idx) {
 int i=0; bool done = false, whole_done=false;
 while (!done && i<N) {
  if (idx[i] == tail_idx[i]-1) {
    idx[i] = head_idx[i];
    if (i == N-1) whole_done = true;
    i++;
  } else {
     idx[i]++; done = true;
  }
 }
  return whole_done;
}
```

Speeding Up N-Dimensional Base Kernel

- Relaxed the constraint that 'N' is not known at program-time, but known at compile-time.
- Created script to generate the base kernel for a given N
- Unroll loop to do 2D at a time

In original base kernel, for each t :

```
while (!done) {
        kernel(Q, t, l_index);
        done = update_index(l_index,

        l_head_index, l_tail_index);
}
```

In original kernel, for each dimension:

```
l_idx[i]++;
tmp += U(Q, t, l_idx);
l_idx[i]--;
tmp += -2.0 * U(Q, t, l_idx);
```

In new base kernel, for each t :

```
for(int x0= l_head_index[0]; x0<l_tail_index[0]; x0++) {
for(int x1= l_head_index[1]; x1<l_tail_index[1]; x1++) {
....
temp_index[0]=x0+1; temp_index[1]=x1;
....
asm volatile(
 "movups (%0), %%xmm0\n\t"
"addps (%1), %%xmm0\n\t"
"subps (%2), %%xmm0\n\t"
"mulps (%3), %%xmm0\n\t"
"movups %%xmm0, (%0)"
::"r"(vec1),  "r"(vec2), "r"(twovec), "r"(cvec)
);
```

With SIMD & Loop Unrolling vs Without

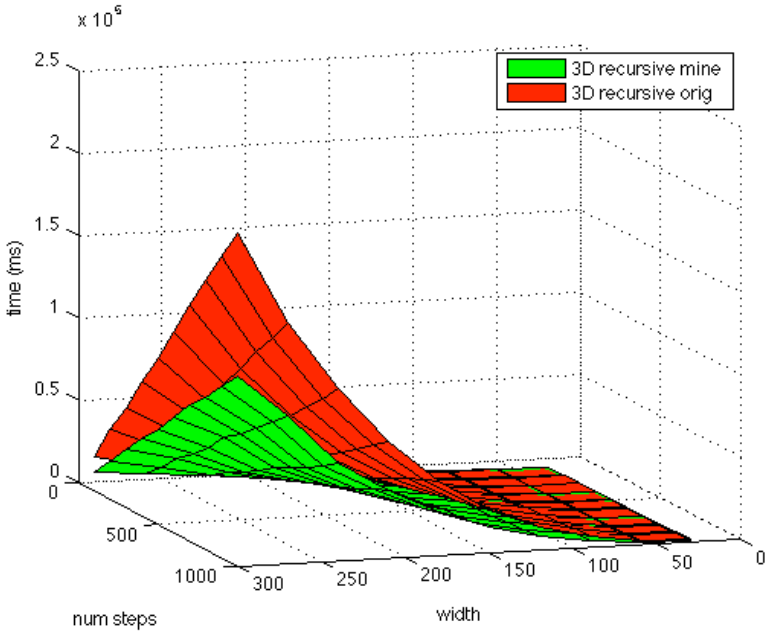
- In 3D both the loops and recursive SIMD implementations show decreases in runtime for smaller widths.
- For loops implementation, speedup is bigger for smaller widths. For recursive implementation, speedup stays relatively constant for all width values.
- Speedup for X=100 : loops 30% recursive 42%
- Speedup for X=300 : loops -- recursive 42%

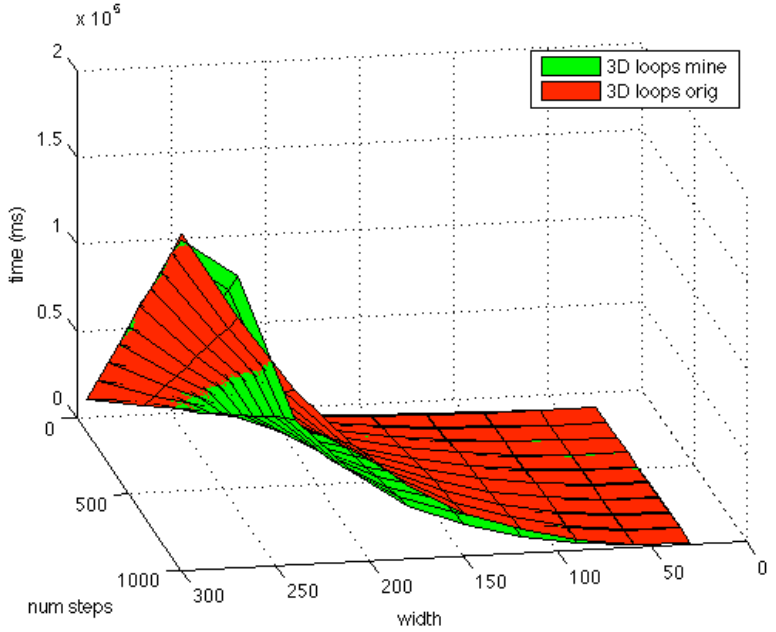
Loops	Without	SIMD+Loop
Range - 50x50 to 300x300	129 ms to 193125 ms	80 ms to 192068 ms
Average	32572 ms	30606 ms

Recursive	Without	SIMD+Loop
Range - 50x50 to 300x300	64 ms to 202751 ms	45 ms to 115570 ms
Average	34091 ms	19862 ms

With SIMD & Loop Unrolling vs Without

Loops	Without	SIMD+Loop
Range - 50x50 to 300x300	129 ms to 193125 ms	80 ms to 192068 ms
Average	32573 ms	30606 ms





Recursive	Without	SIMD+Loop
Range - 50x50 to 300x300	64 ms to 202751 ms	45 ms to 115570 ms
Average	34091 ms	19862 ms

Speeding Up N-Dimensional Base Kernel: SIMD vs Loop Unrolling

- Adding in just SIMD instructions, produces speedup of 13%
- Adding in both SIMD and loop unrolling, produces speedup of 42%

Recursive	Without	SIMD	SIMD+Loop
Range - 50x50 to 300x300	64 ms to 202751 ms	58 ms to 174736 ms	45 ms to 115570 ms
Average	34091 ms	29551 ms	19862 ms

Boundary Conditions – Original

- Conditional for every point to determine if on boundary.
- Original Kernel:

```
if (x == 0 || x == Q->X-1 || y == 0 || y == Q->Y-1) {
    U(Q, t+1,x,y) = IDENTITY;
}
else
   U(Q, t+1,x,y) = Q->CX * (U(Q, t,x+1,y) - 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
   + Q->CY * (U(Q, t,x,y+1) - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
   + U(Q, t,x,y);
```

Boundary Conditions – Bitmap

- Pre-compute bitmap, one bit per point
  - If boundary, bit = 0
  - Else, bit = 1
- ~17% faster
- Bitmap Kernel:

```
U(Q, t+1,x,y) = (B(x,y) ? Q->CX * (U(Q, t,x+1,y) - 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
+ Q->CY * (U(Q, t,x,y+1) - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
+ U(Q, t,x,y) : U(Q,t,x,y));
```

Boundary Conditions – Zero-padding

- In base case computation
  - Initialize boundaries to be Identity. Only call kernel on non-boundaries.
  - ~25% faster
  - Zero-padding
    - Extra storage to save IDENTITY elements (non-periodic boundary)
    - Storage overhead:

$$(size0 + 2 * slope0)$$
$$\times(size1 + 2 * slope1) \times \ldots$$
$$\times(sizeN + 2 * slopeN)$$
$$-(size0 \times size1 \times \ldots \times sizeN)$$

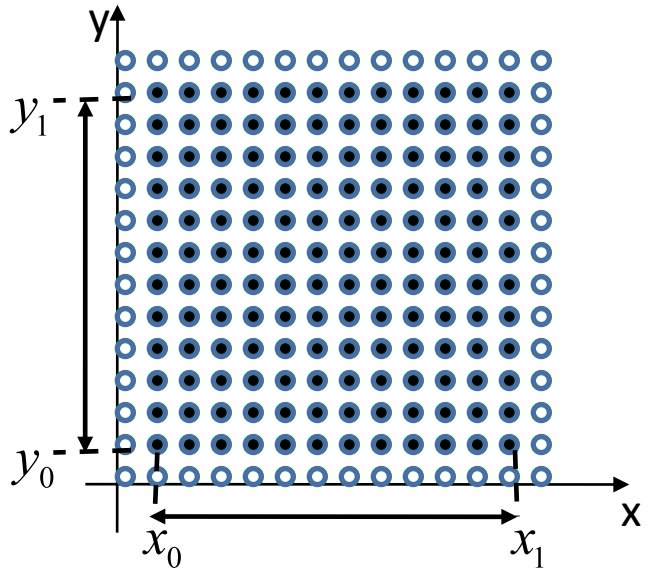
    - Assuming:

$$size0 = size1 = \ldots = sizeN$$
$$slope0 = slope1 = \ldots = slopeN$$

    - Overhead :

$$(size + 2 * slope)^{N+1} - size^{N+1}$$

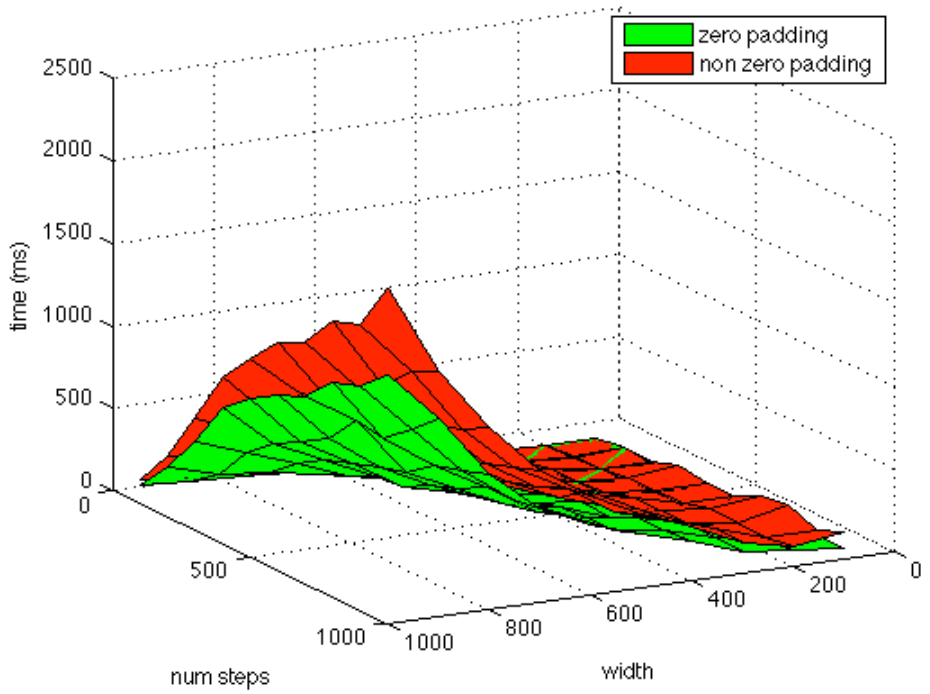
```
U(Q, t+1,x,y) = Q->CX * (U(Q, t,x+1,y)
- 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
+ Q->CY * (U(Q, t,x,y+1) - 2.0 * U(Q,
t,x,y) + U(Q, t,x,y-1))
+ U(Q, t,x,y);
```



Boundary Conditions

- Speedup of zero-padding over non zero-padding (around 25% improvement, depending on the size of grid, the larger the size, the larger the performance gains)

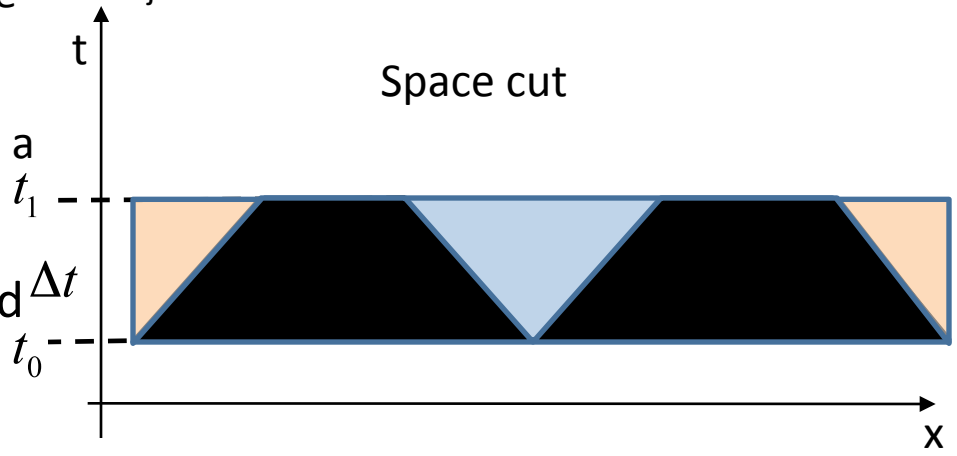
	Zero-padding	Original
Range - 100x100 to 1000x1000	6 ms to 1500 ms	10 ms to 2021 ms
Average	315 ms	418 ms



Periodic versus Non-Periodic boundaries

- In non-periodic boundary, we can utilize zero-padding technique to greatly reduce the overhead accessing/updating of boundary element.
- In periodic boundary, we 'have to' adopt the modulo operations in base case calculation
  - Accumulated cost is very expensive
  - Reduce the benefits of recursive algorithm compared against loop algorithm ---- modulo op becomes a dominant performance penalty
- We have to merge the beginning and end element into one trapezoid to avoid update of first and last element at different time steps.

```
If (l_start==0 && l_end = size[i]) {
    l_grid.x0[i] = l_end;
    l_grid.dx0[i]=-slope[i];
    l_grid.x1[i] = l_end;
    l_grid.dx1[i] = slope[i];
    cilk_spawn walk(t0, t1, l_grid);
} else {
    walk(beginning triangle);
    walk(end triangle);
}
```



Irregular Shaped Domain

- Motivation for irregular boundaries:
  - We may be concerned with running simulations on more than a simple grid.
  - i.e. calculating diffusion of some molecules in a cell over time.

Irregular Shaped Domain

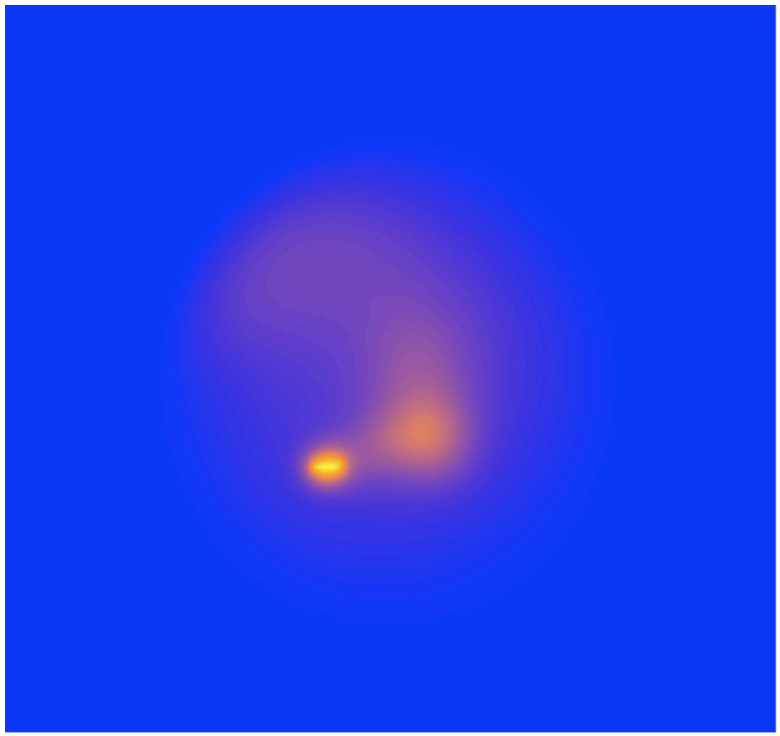
- Use Bitmap for Implementation:

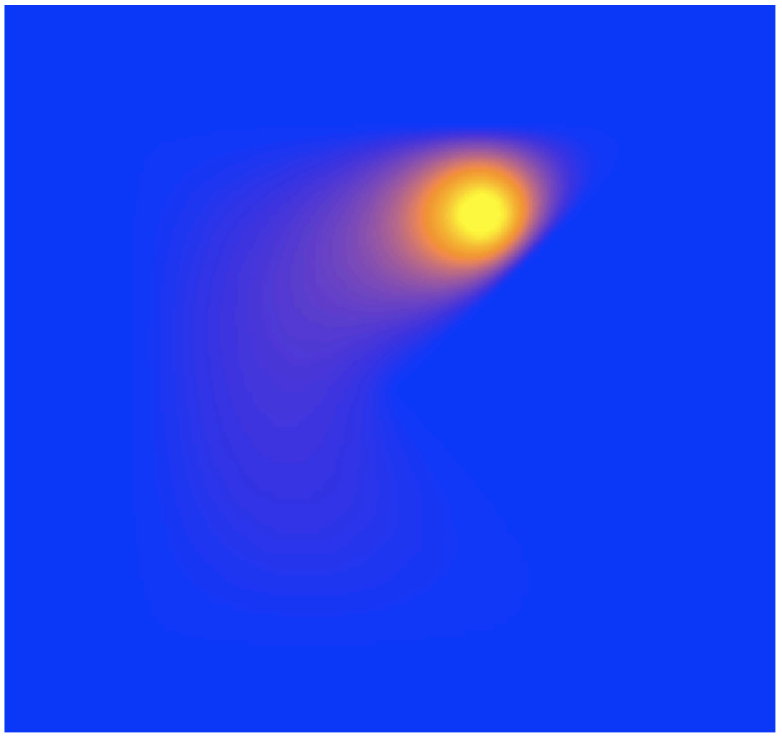
```
U(Q, t+1,x,y) = (B(x,y) ? Q->CX * (U(Q, t,x+1,y) - 2.0 * U(Q, t,x,y) + U(Q,
t,x-1,y))
+ Q->CY * (U(Q, t,x,y+1) - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
+ U(Q, t,x,y) : U(Q,t,x,y));
```

- Can manually enter the boundaries into bitmap, or use one of our helper functions.
  - polyBoundary(...
  - circBoundary(...

Irregular Shaped Domain

- Demonstration

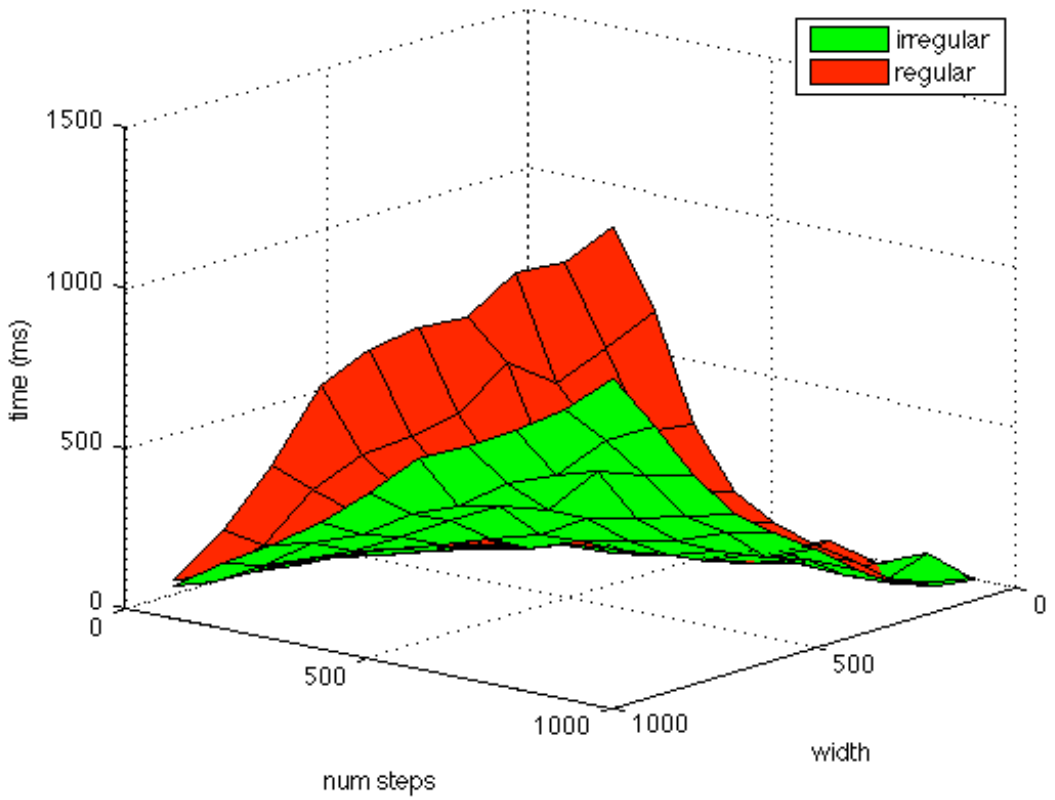




Irregular Shaped Domain

26% faster on average, depends on effective area of irregular grid

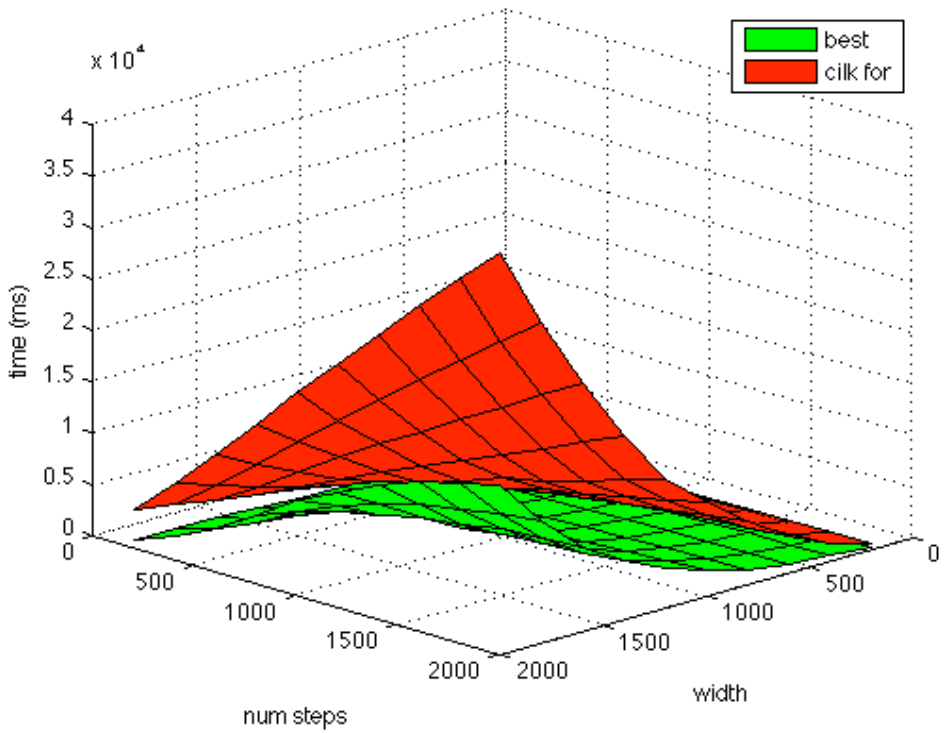
	Regular	Irregular
Range - 100x100 to 1000x1000	6 ms to 1500 ms	6 ms to 1026 ms
Average	315.0 ms	234.6 ms



Loops vs Our Best

- Speed of simple parallelization by cilk_for vs speed of our best algorithm (not including SIMD + Loop Unroll). As width and num steps increase, so do the gains from optimizations.
- 2.6 times faster

	cilk_for	Our best
Range - 200x200 to 2000x2000	52 ms to 39042 ms	15 ms to 16508 ms
Average	8088 ms	3127 ms



Other Optimizations

- Non-Effective
  - Static Elaboration
  - Always Cut Into Largest Spatial Dimension
  - Overlap (on large grids)

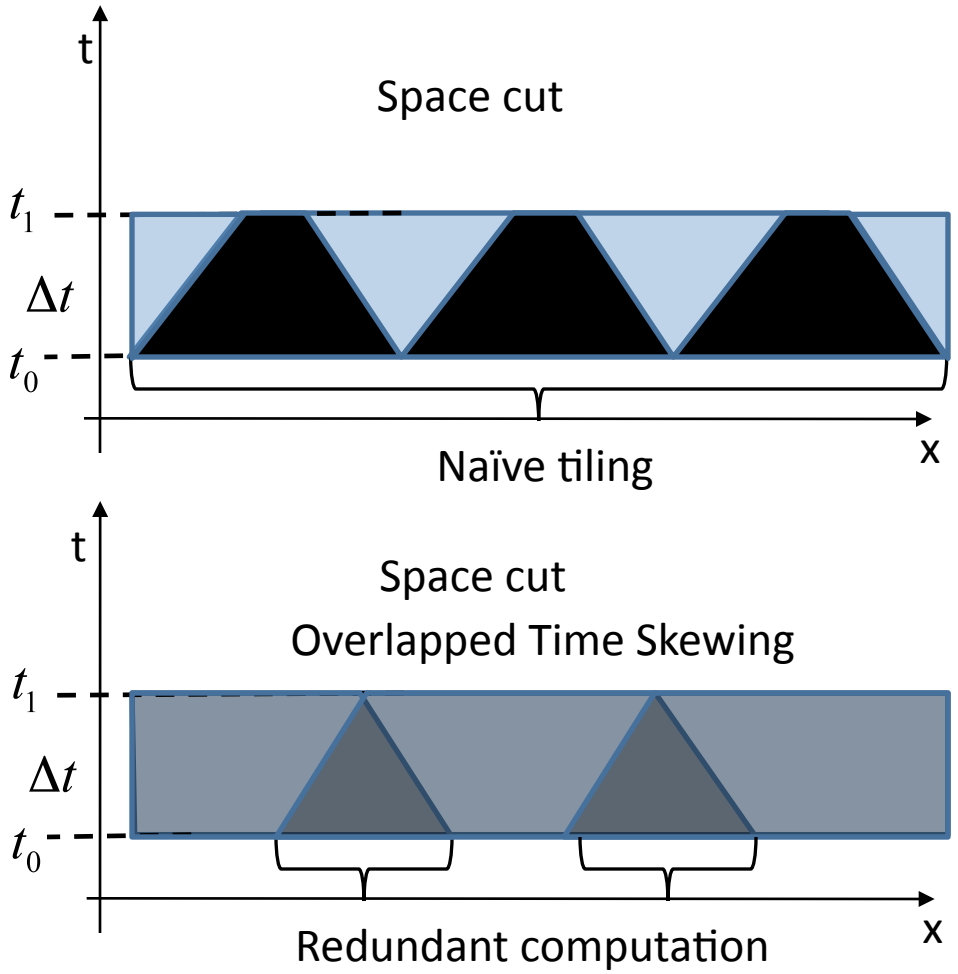
Next steps

- Further improve the performance by refining the base case computation
  - Expression template (may call some existing libraries, such as Blitz++, freepooma, etc.)
- Optimizing the arbitrary N-dimensional Stencil
- Figuring out some real world applications of stencil computation on irregular shaped grids and develop some optimization for that.
- Define user specification
- Apply to other stencils, such as Lattice Boltzmann

Backup Slides

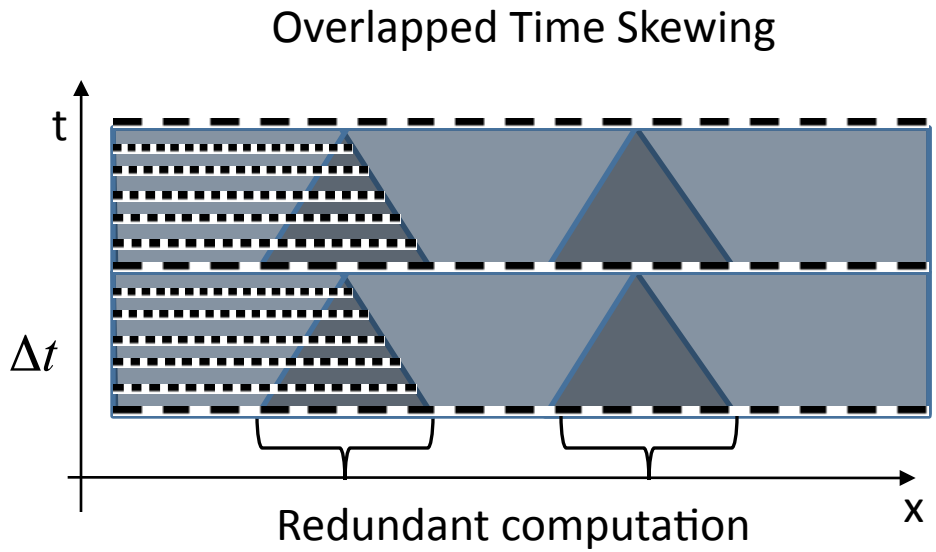
Faster : Overlapped Time Skewing

- Cutting heuristics for cache-oblivious algorithm:
  - In each space cut, we adopt Naïve Tiling strategy.
  - In each space cut, we adopt Overlapped Time Skewing strategy
    - Big toggle array + small toggle array



Faster : Overlapped Time Skewing

- Large toggle array for time cuts t0 t1 t2 etc...
- Smaller toggle array per trapezoid within t0 → t1, t1 → t2 etc... intervals.



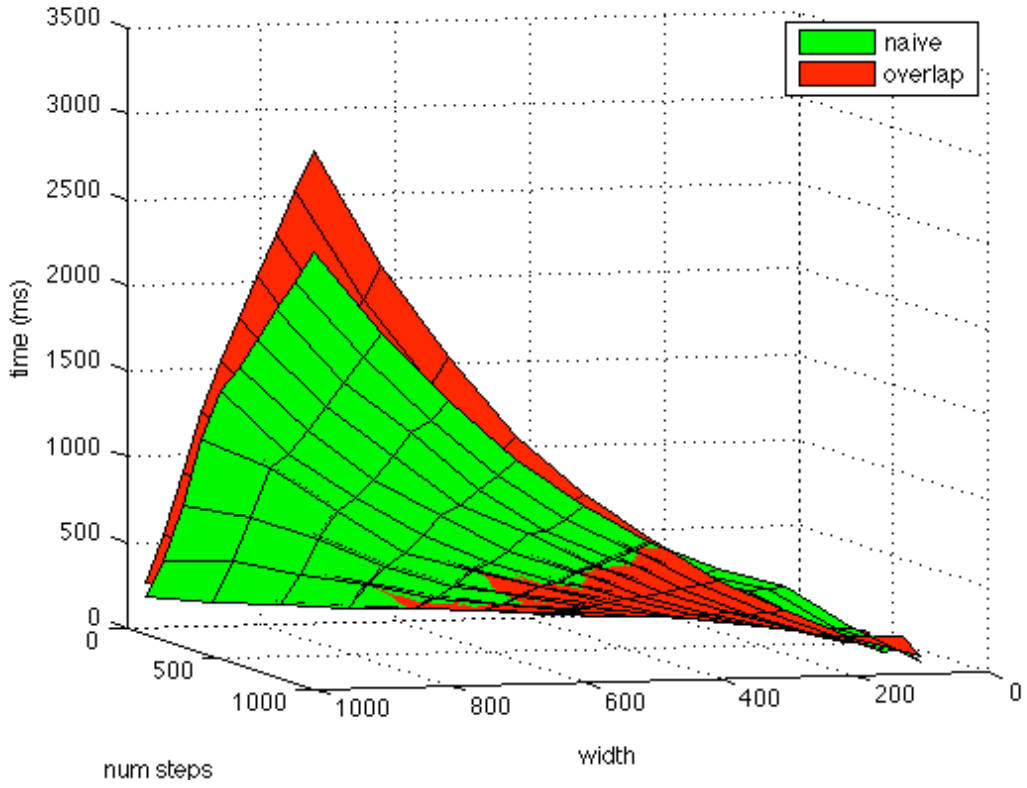
Faster : Overlapped Time Skewing

- Might this be faster?
  - Less burdened parallelism
  - Reduced communication overhead (maybe)
- Might this be slower?
  - Redundant computation
  - More cache misses due to additional toggle array
  - More work per index into small toggle arrays

Faster : Overlapped Time Skewing

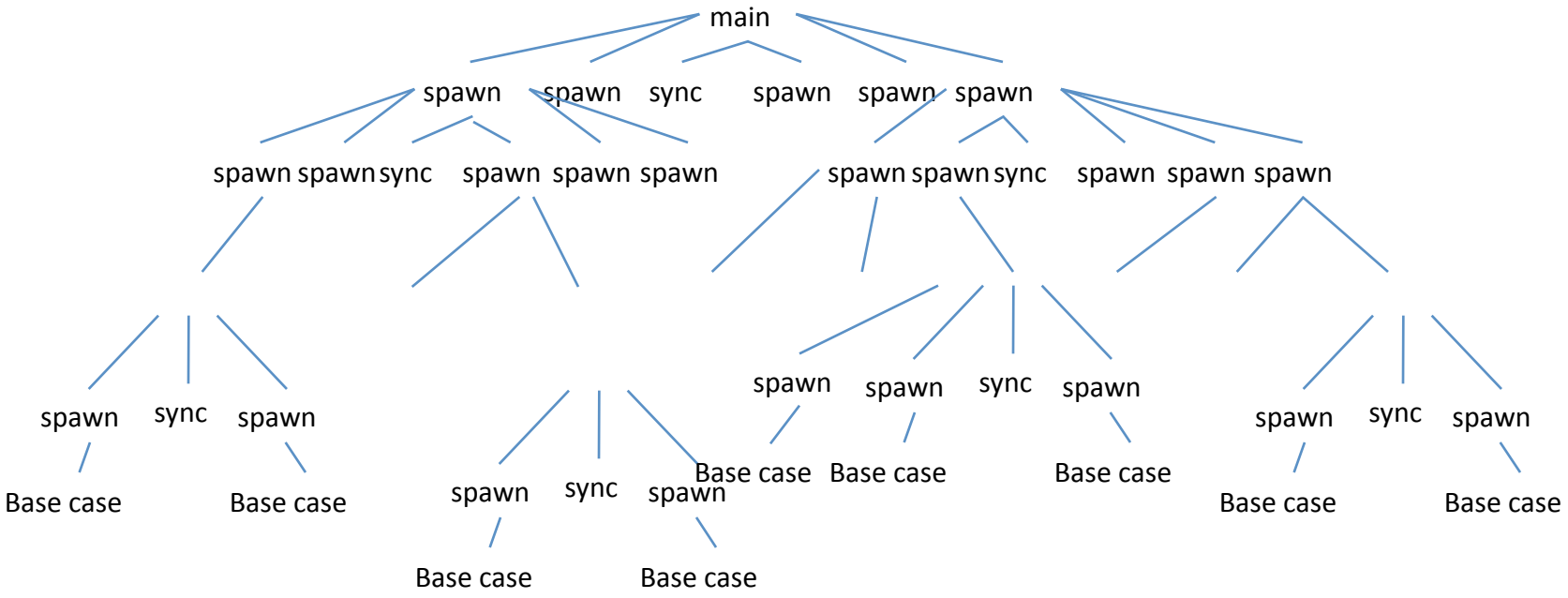
- Faster for smaller grids (less than 500x500)

	Naive	Overlap
Range - 100x100 to 1000x1000	6 ms to 2551 ms	13 ms to 3139 ms
Average	628 ms	684 ms



Faster : Static elaboration

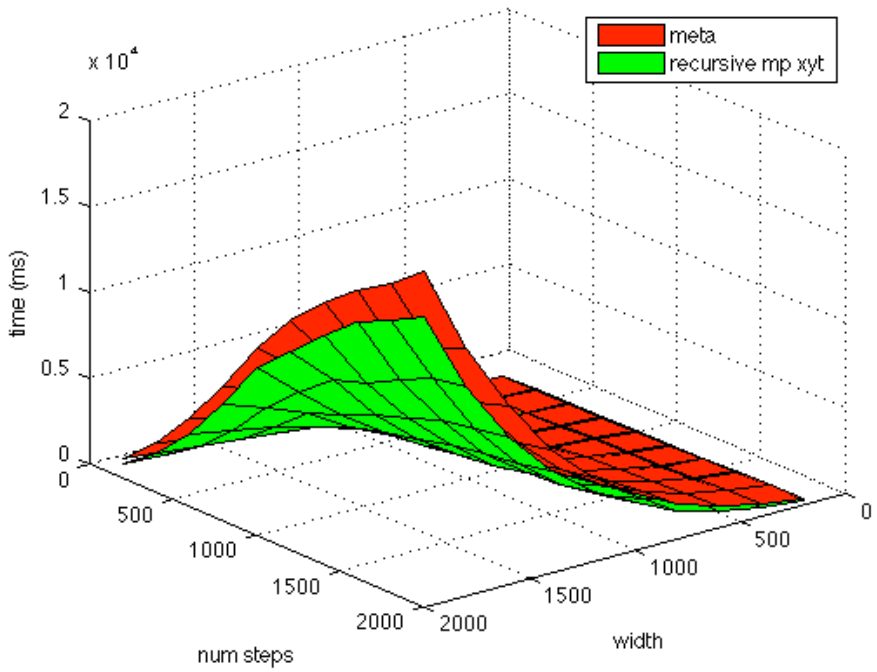
- Motivation: too many recursive function call
- Effort: Elimination by static unroll all the recursive function calls by pre-computing the execution pattern
- Problem: How to efficiently execute the pattern tree?



Static Elaboration vs Our Best

- Just traverse the tree from top down?
- Entire traverse performs worse consistently.

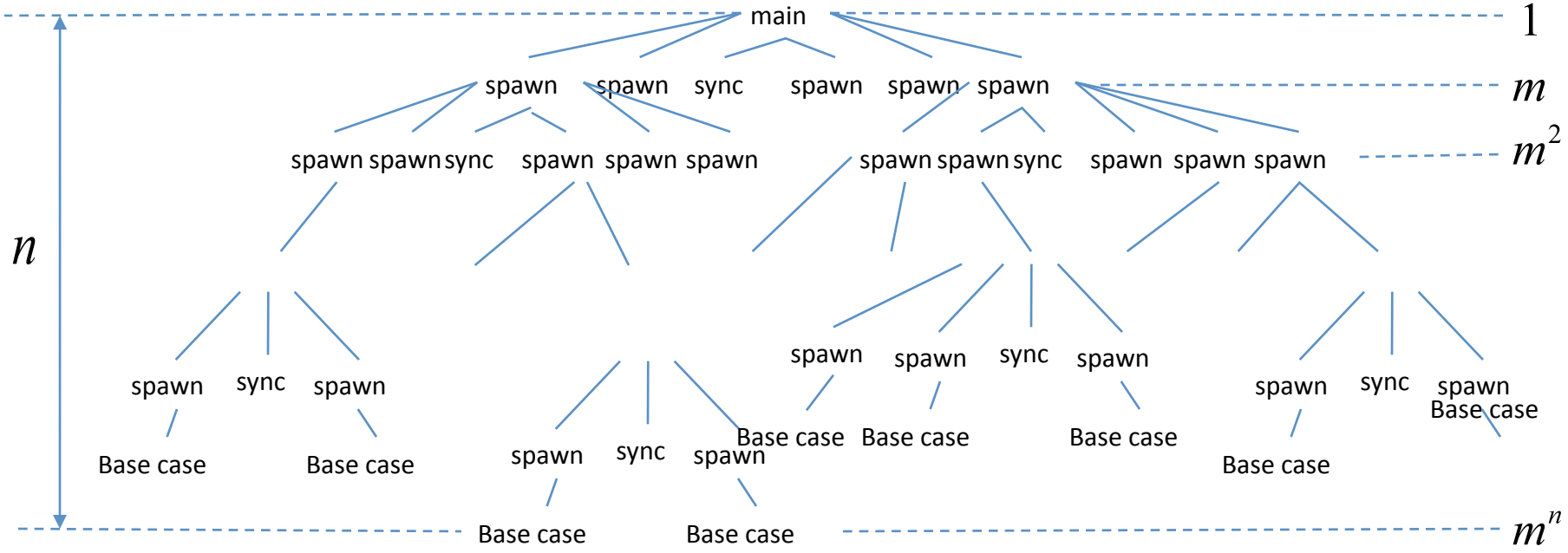
	Static Elaborate	Recursive mp xyt
Range - 200x200 to 2000x2000	33ms to 19534ms	20ms to 16960ms
Average	3801 ms	3236 ms



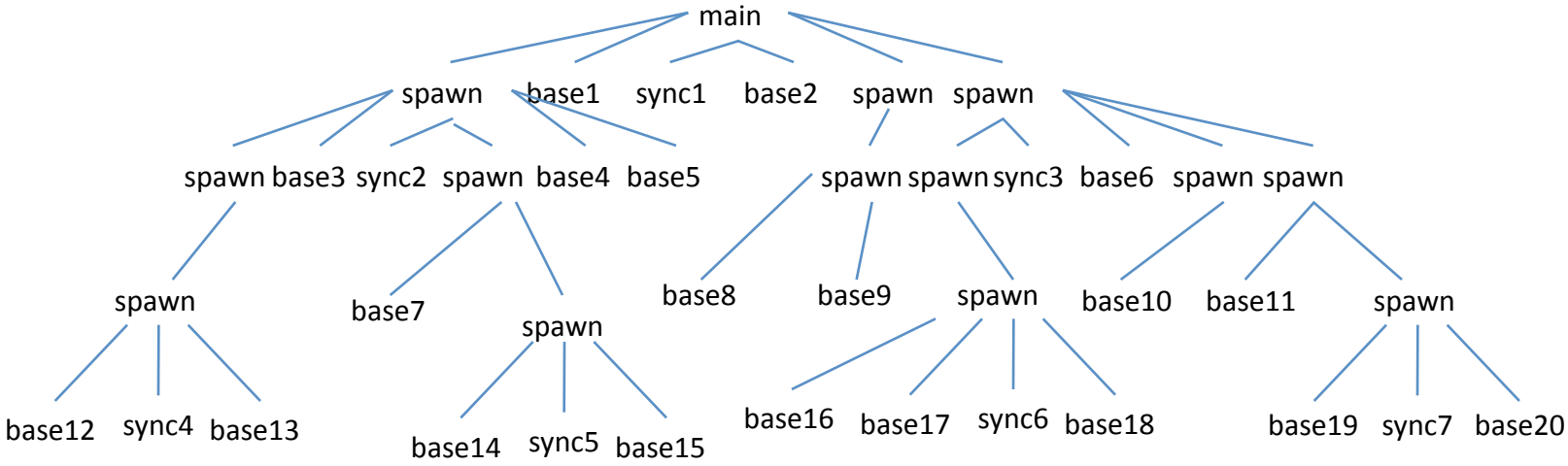
Static elaboration

Lesson: indirect memory access is Very expensive (3801 vs 3348)

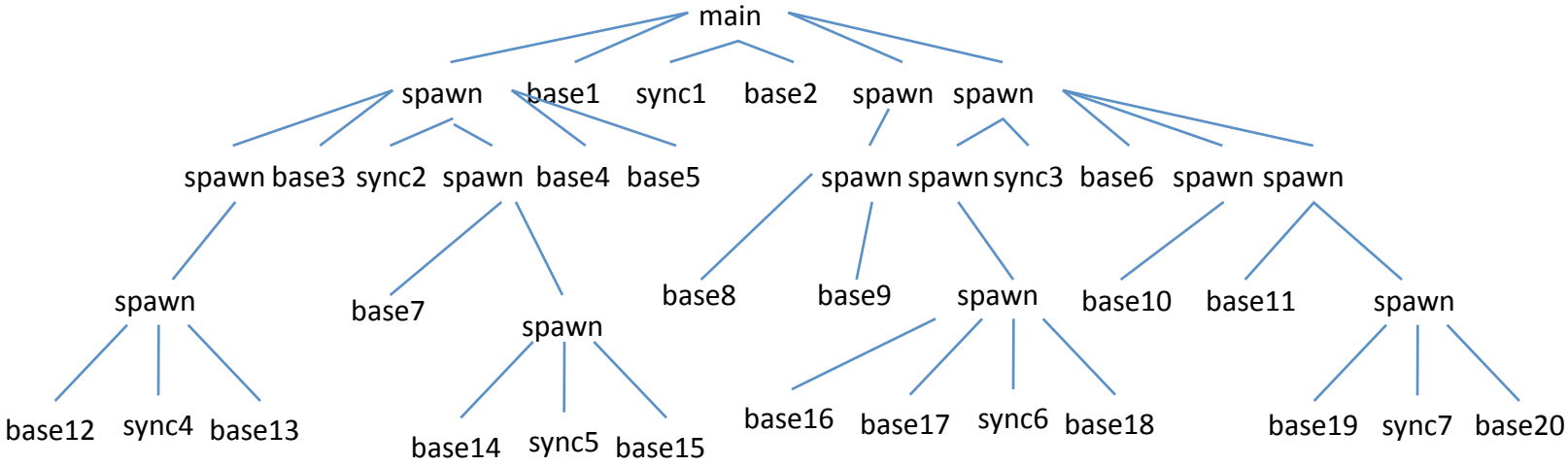
- Second attempt: recursion is just a re-structuring of computing order among base cases.
- The fundamental difference between loop algorithm and recursive algorithm is the computing order of base cases and synchronizations.
- Base cases and synchronizations are the leaves
- Problem: how to execute the leaves ***Only***? $m^n / (1 + m + m^2 + \ldots + m^n)$



Static elaboration

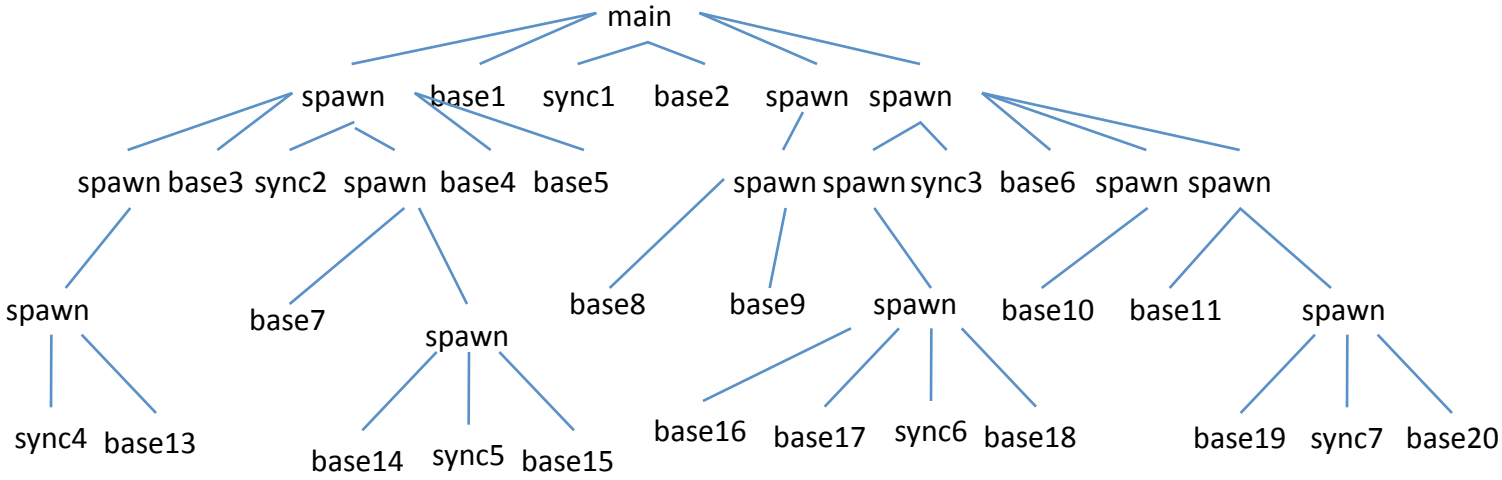


Static elaboration



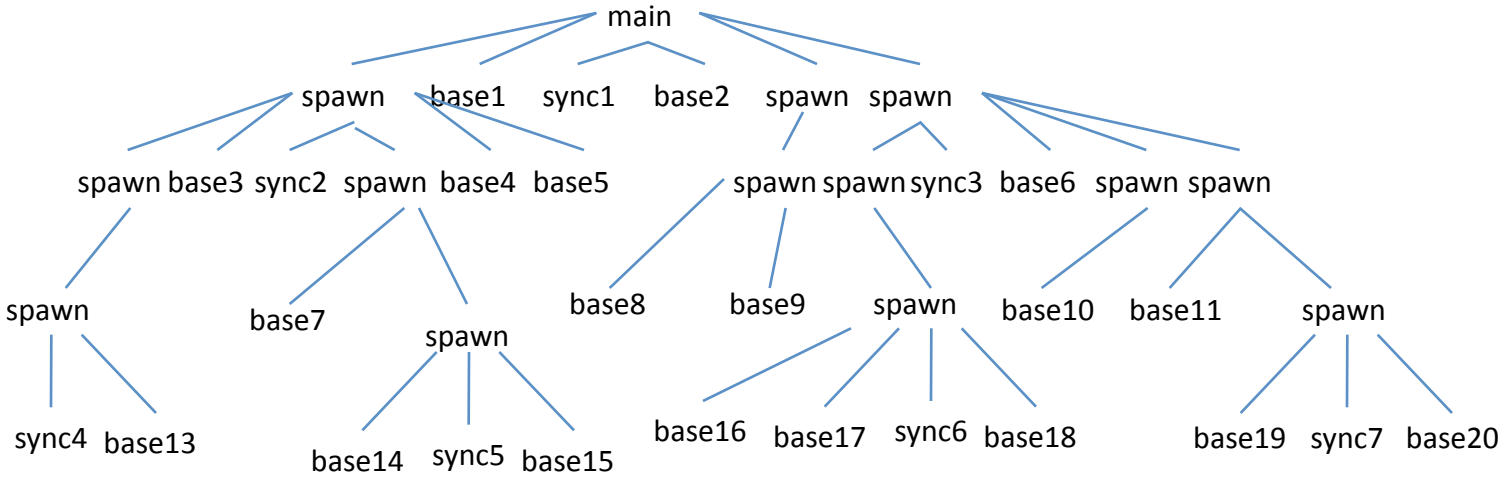
Spawn base12;

Static elaboration



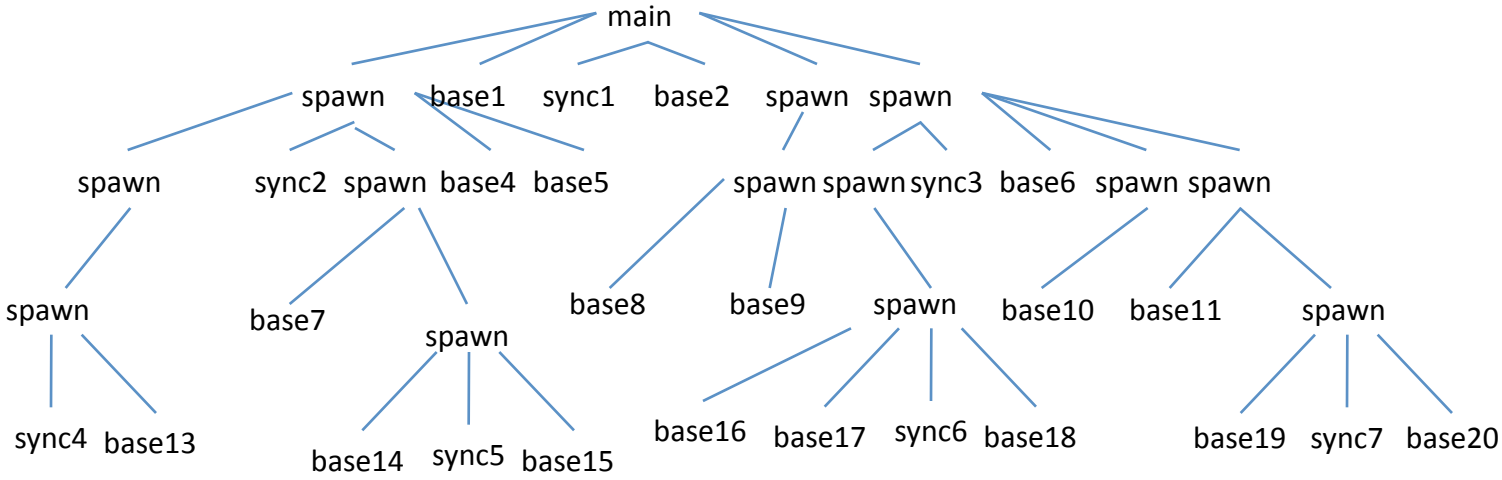
Spawn base12;

Static elaboration



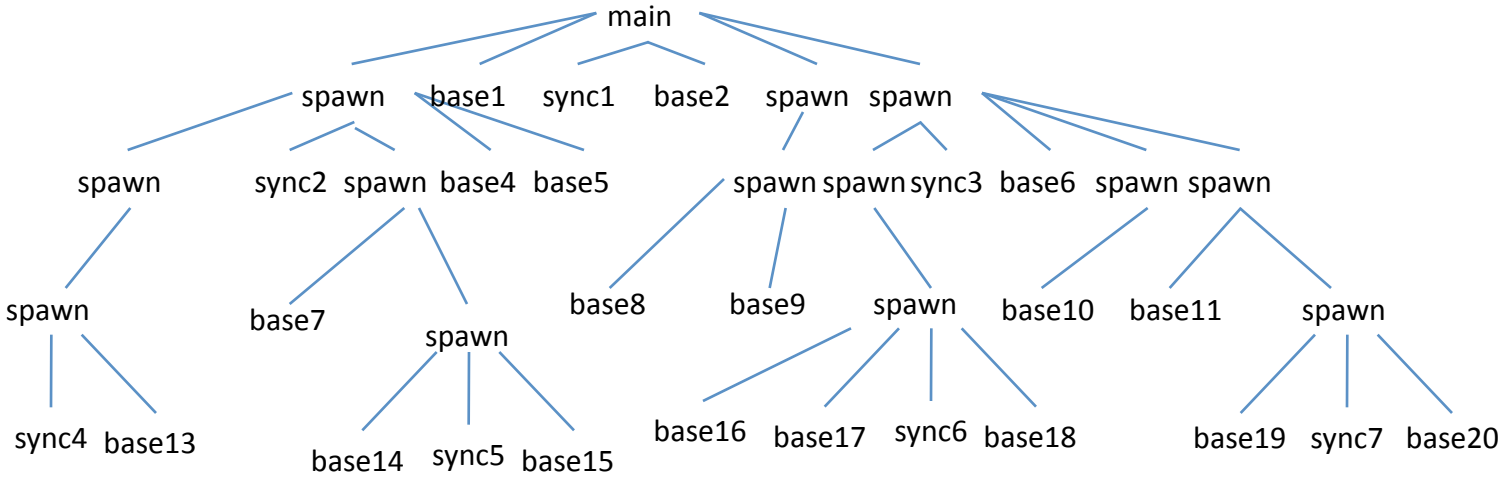
Spawn base12; spawn base3;

Static elaboration



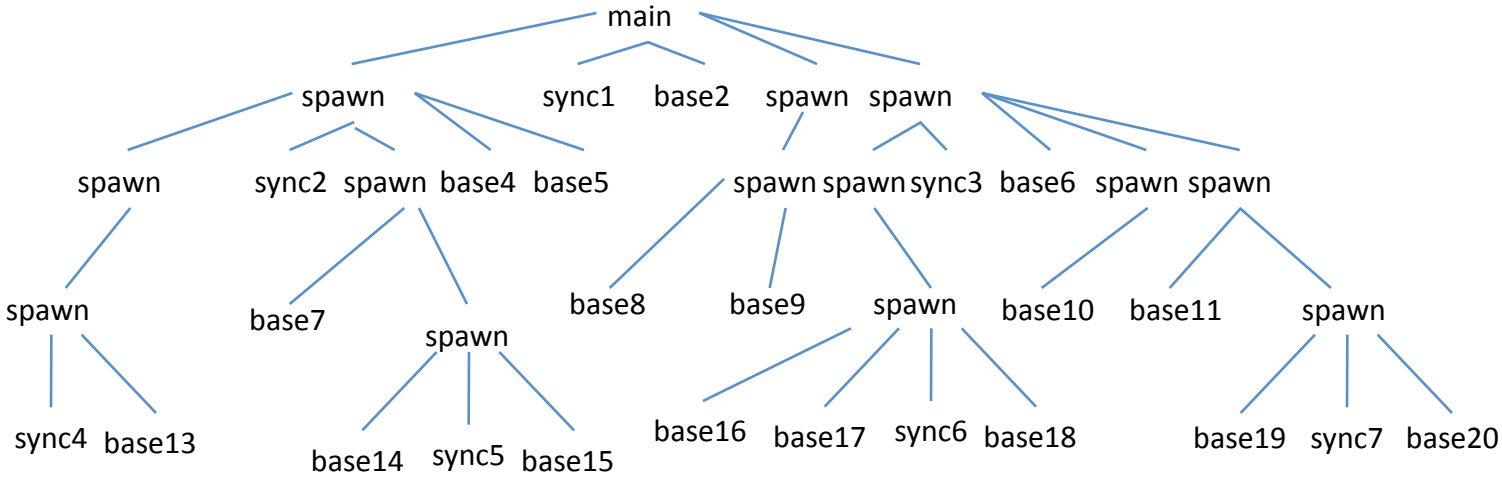
Spawn base12; spawn base3;

Static elaboration



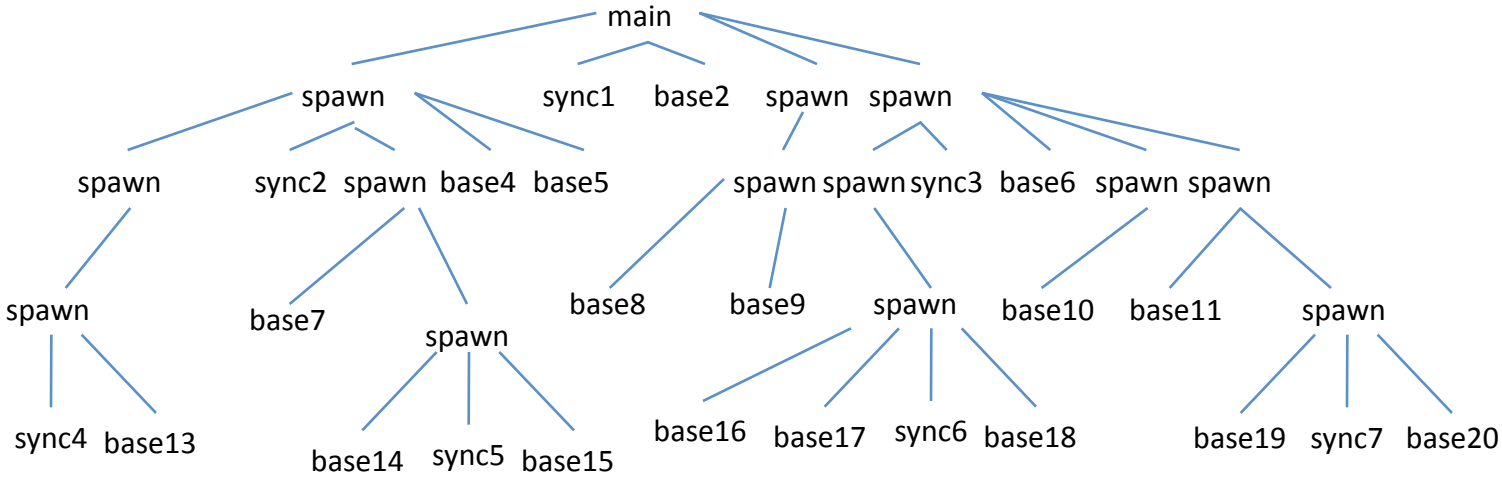
Spawn base12; spawn base3; spawn base1;

Static elaboration



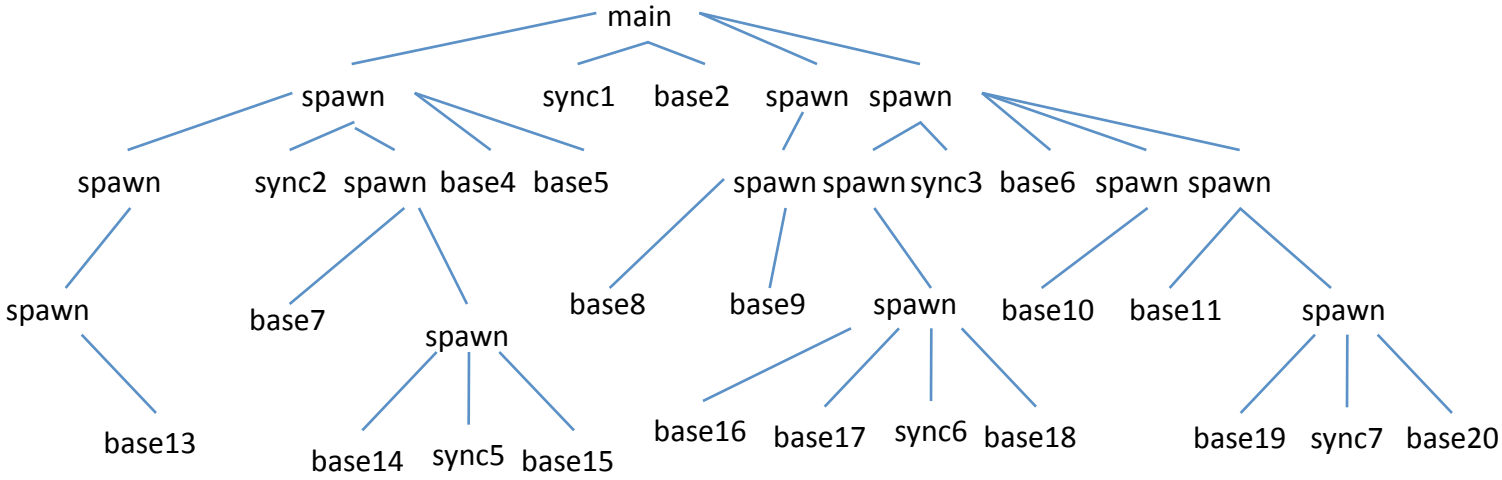
Spawn base12; spawn base3; spawn base1;

Static elaboration



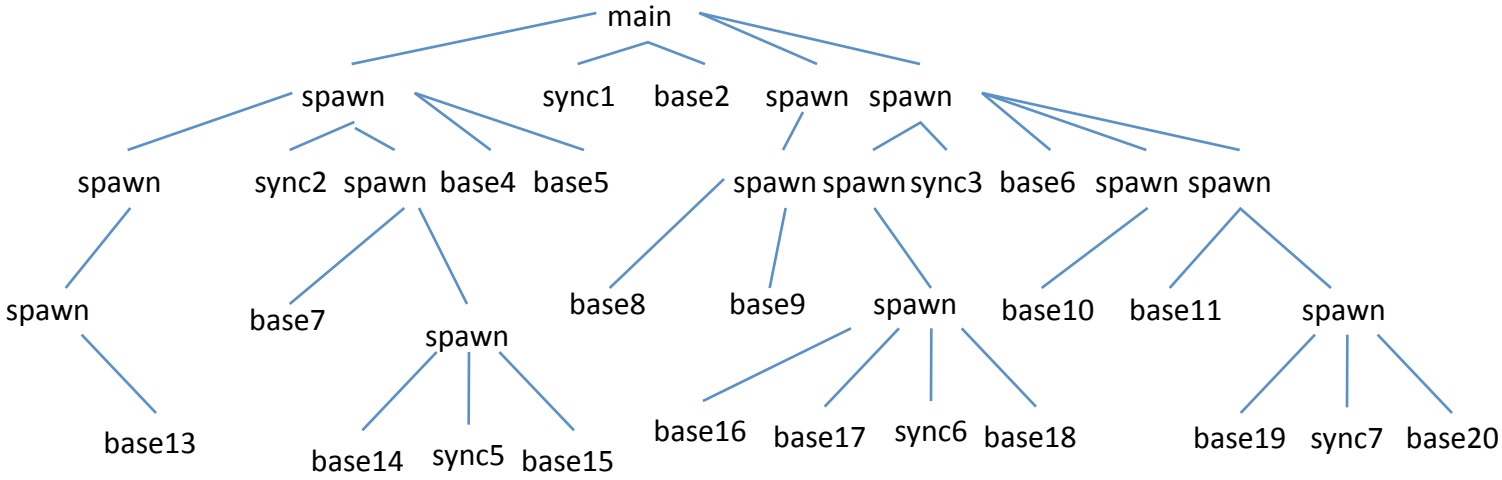
Spawn base12; spawn base3; spawn base1;
Sync4;

Static elaboration



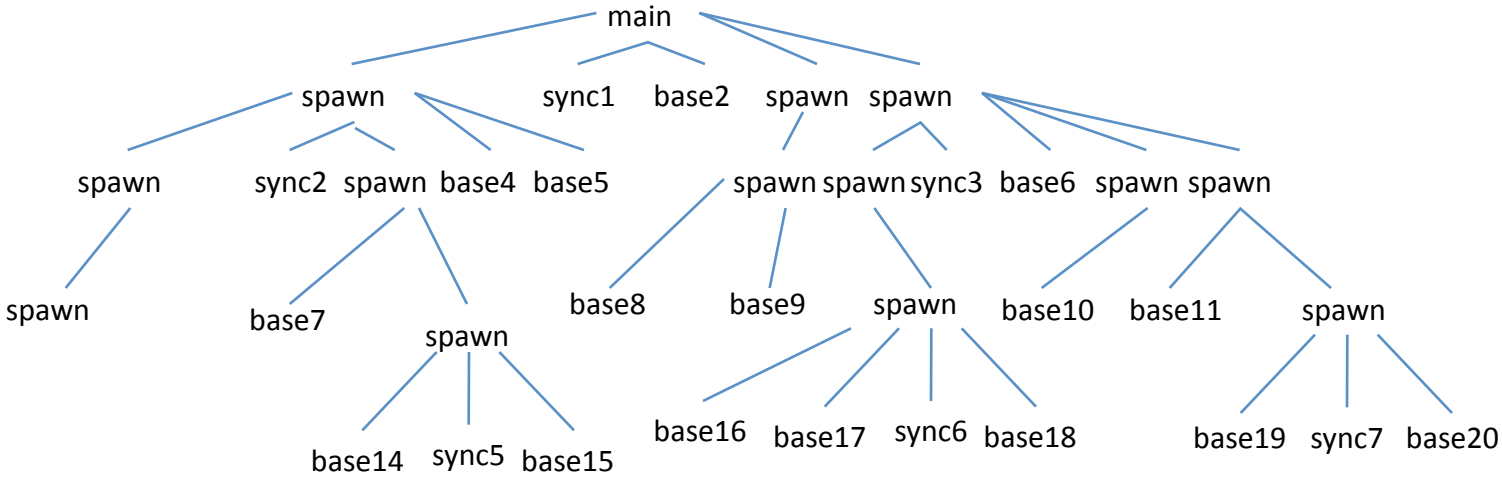
Spawn base12; spawn base3; spawn base1;
Sync4;

Static elaboration



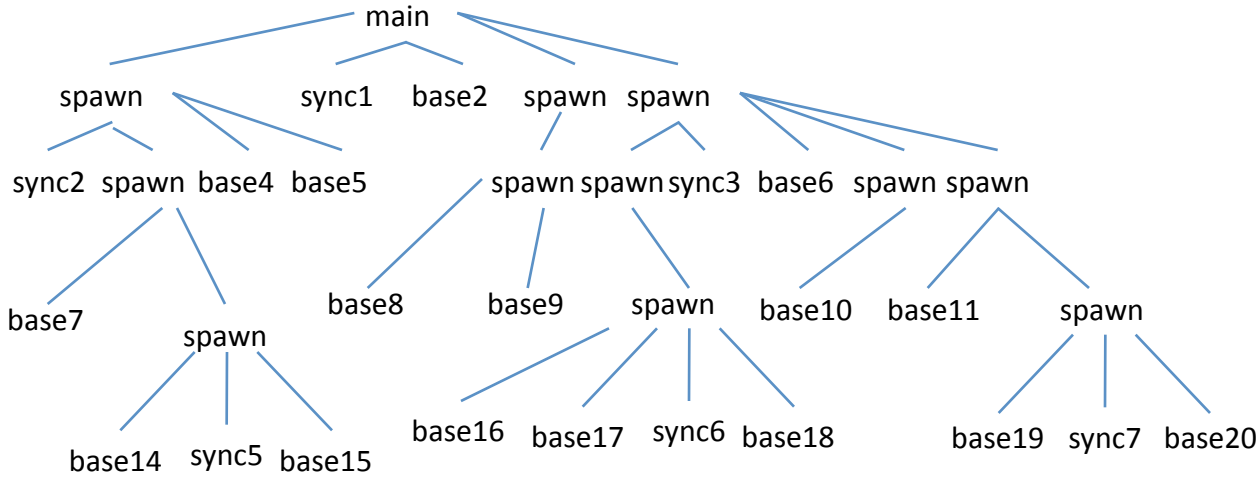
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13;

Static elaboration



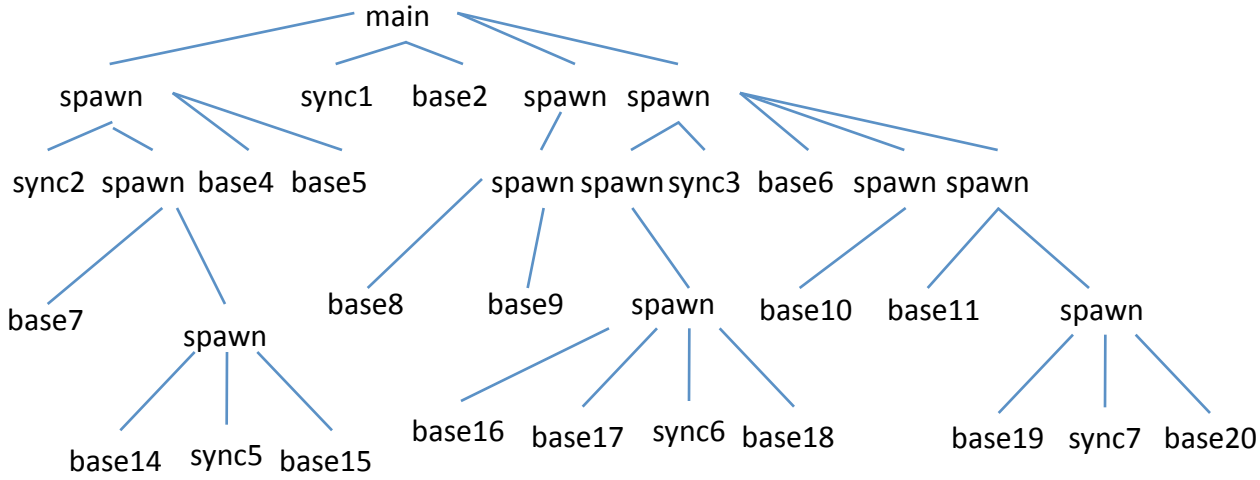
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13;

Static elaboration



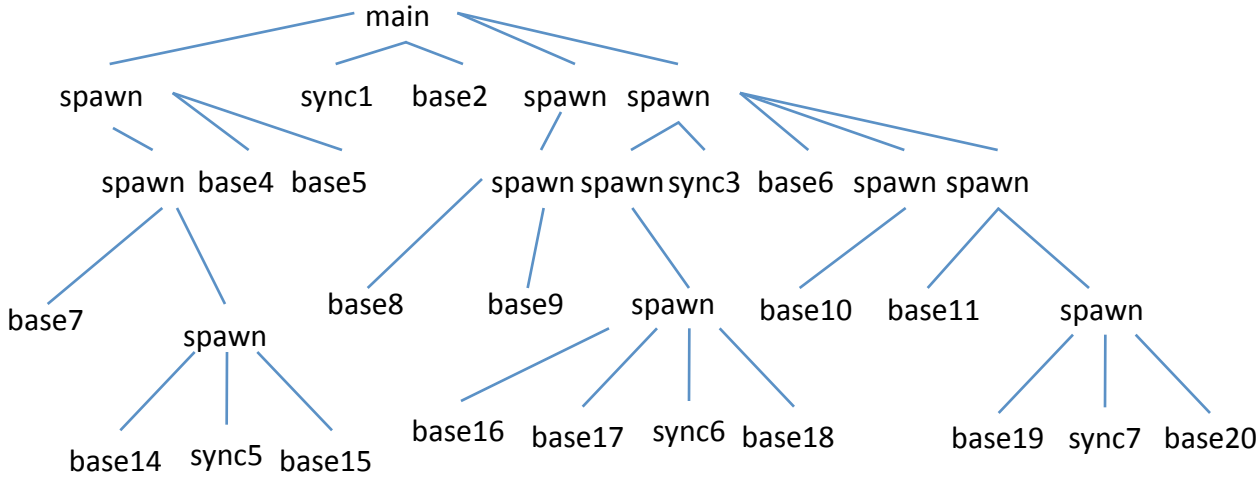
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13;

Static elaboration



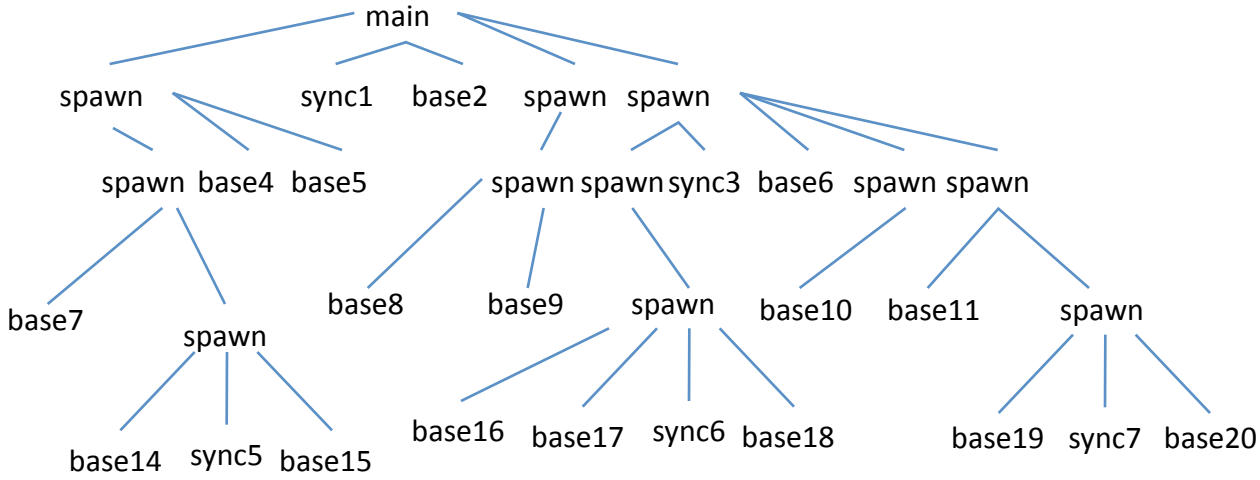
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2;

Static elaboration



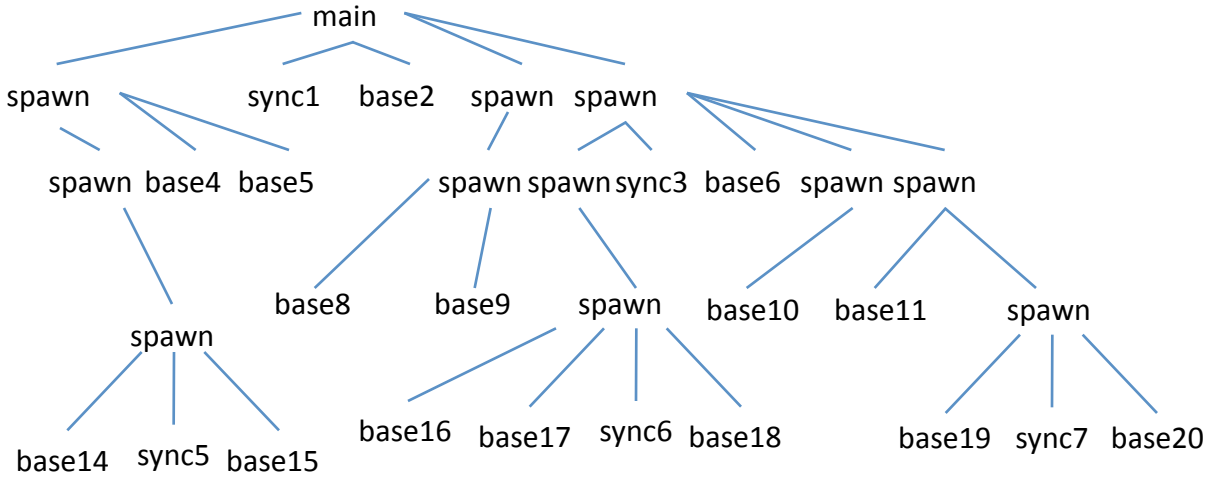
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2;

Static elaboration



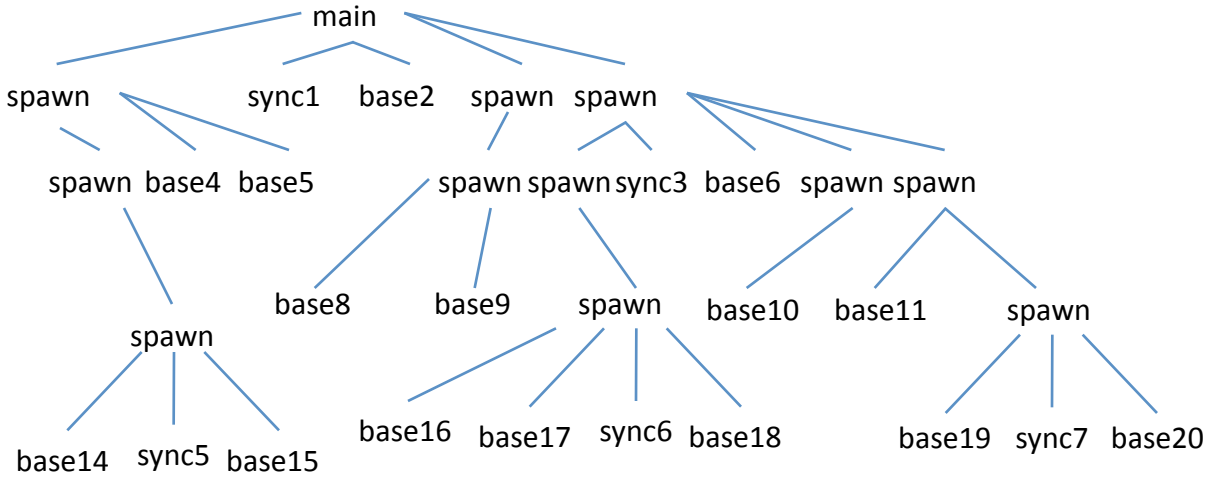
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;

Static elaboration



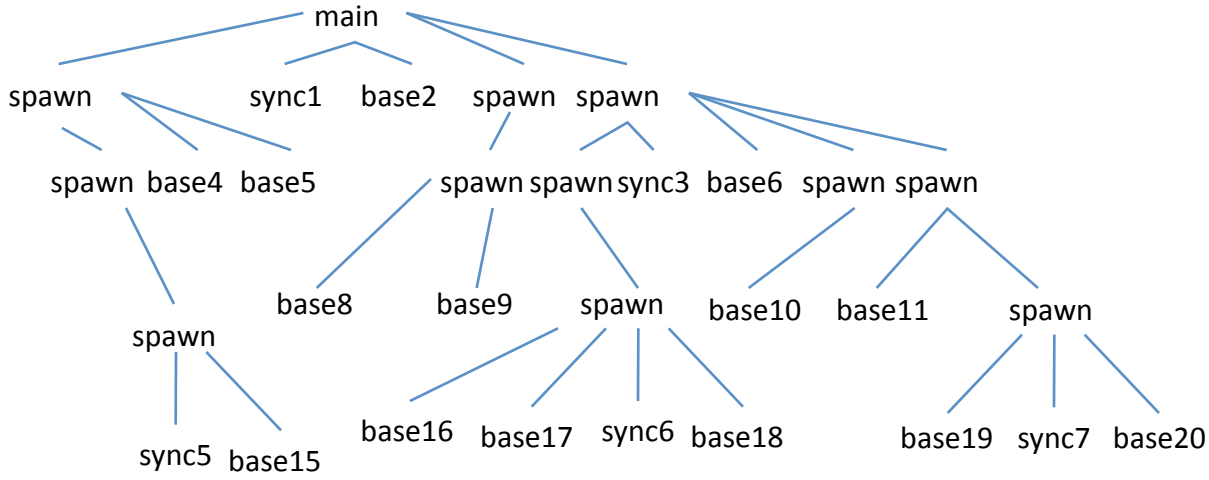
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;

Static elaboration



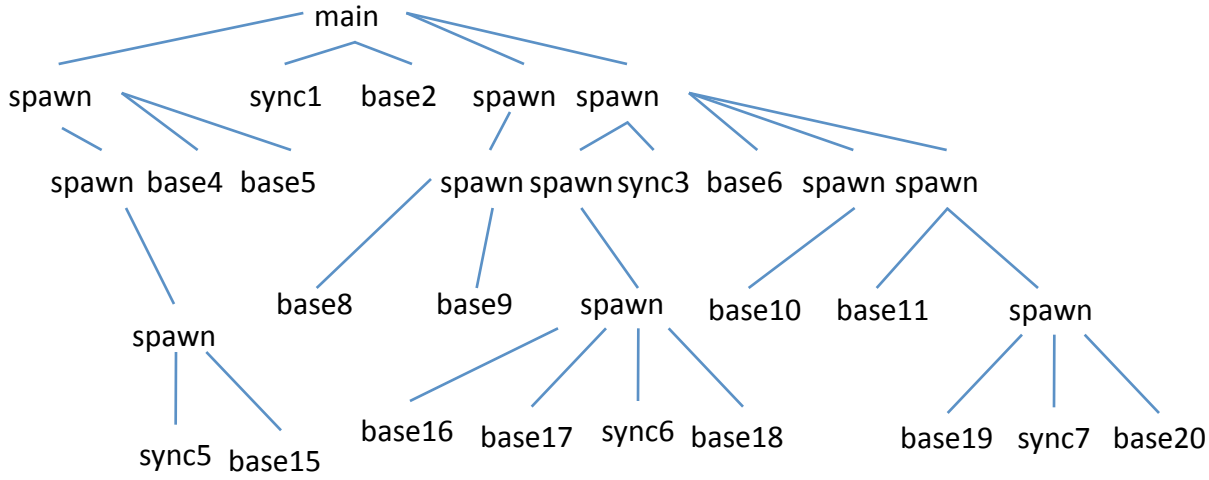
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;

Static elaboration



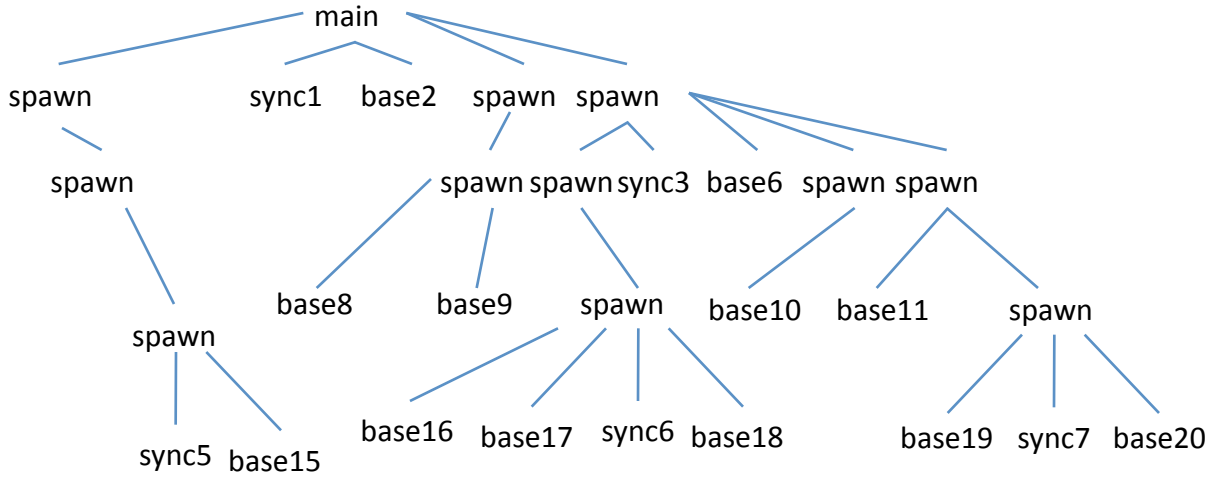
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;

Static elaboration



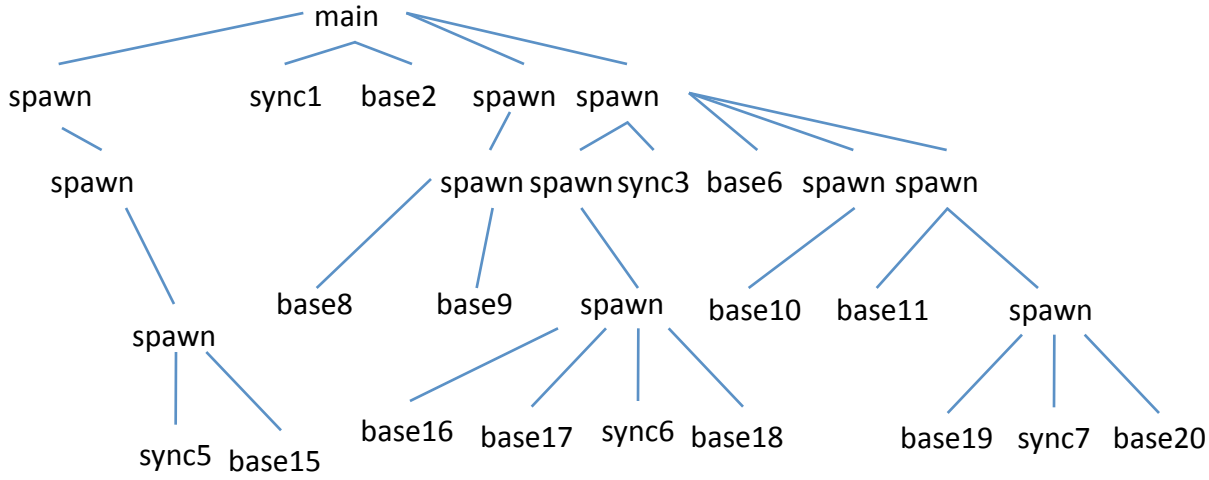
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5;

Static elaboration



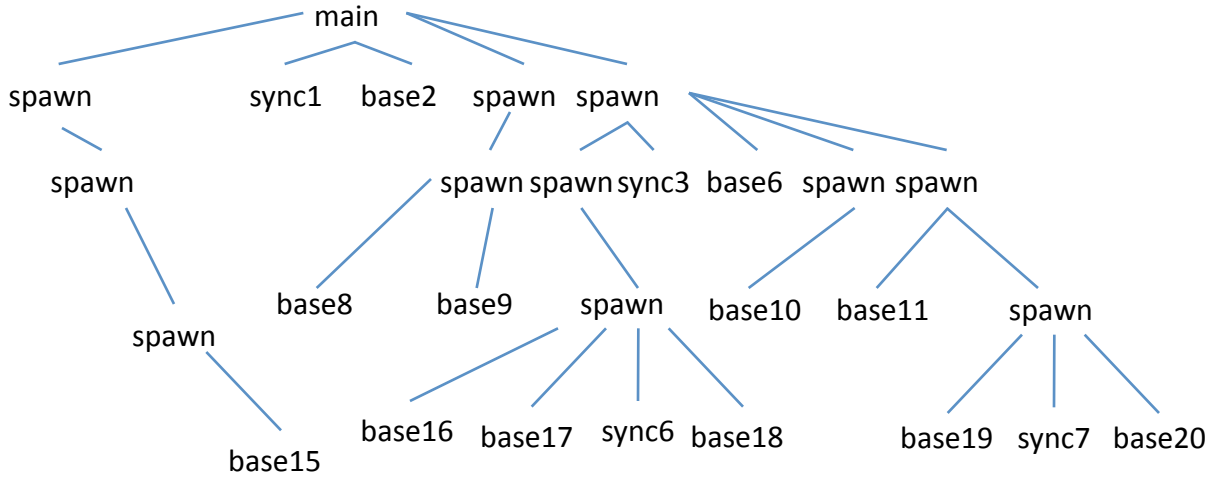
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5;

Static elaboration



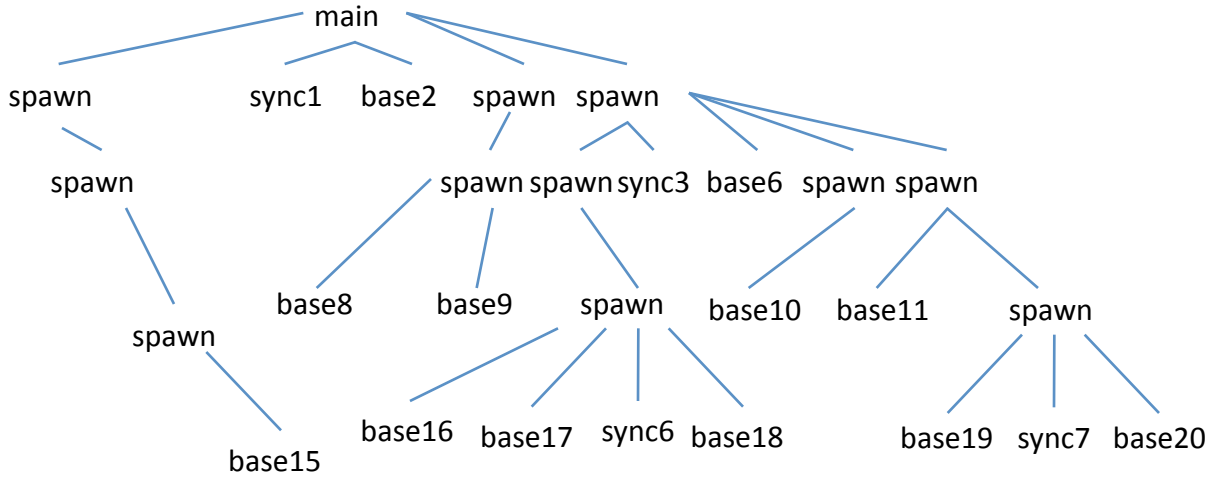
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5;

Static elaboration



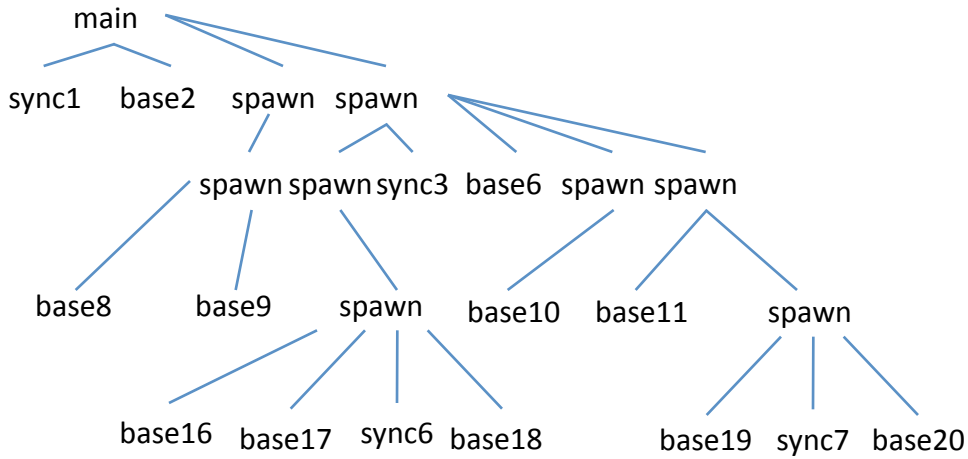
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5;

Static elaboration



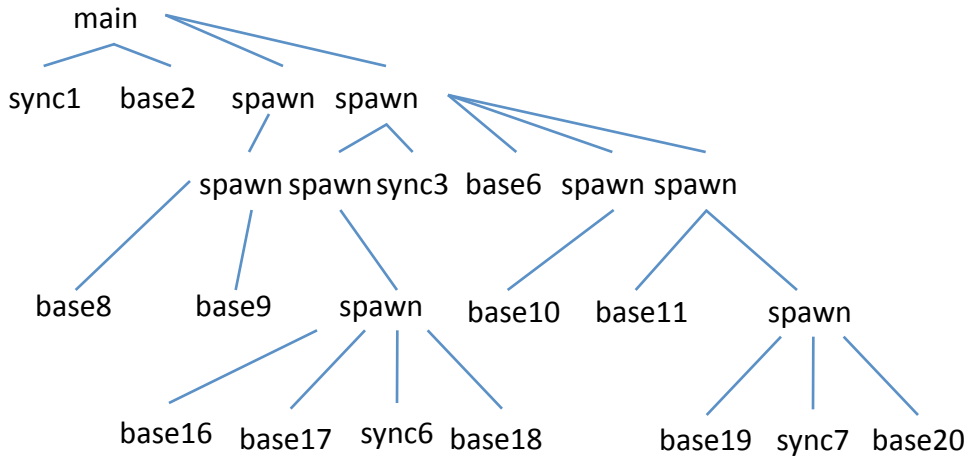
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15;

Static elaboration



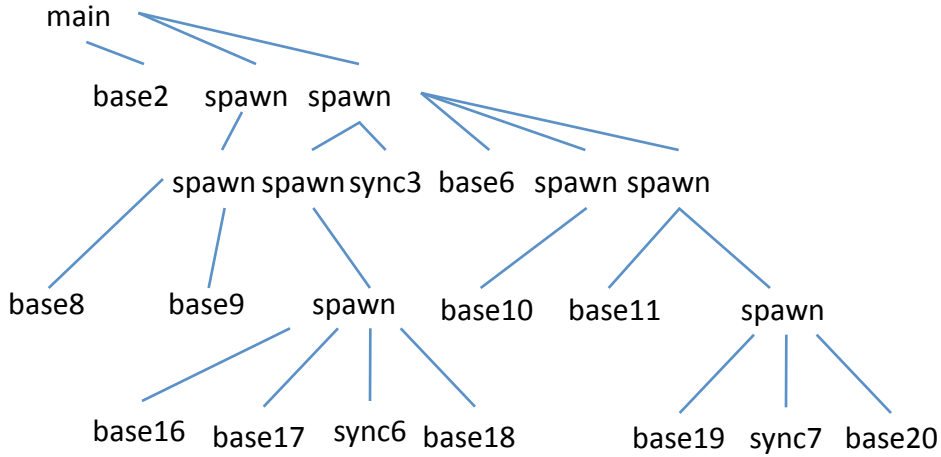
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15;

Static elaboration



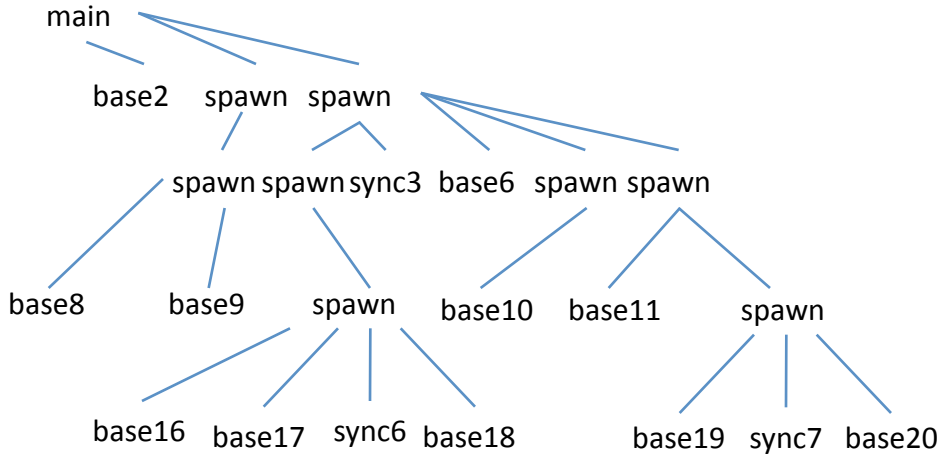
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;

Static elaboration



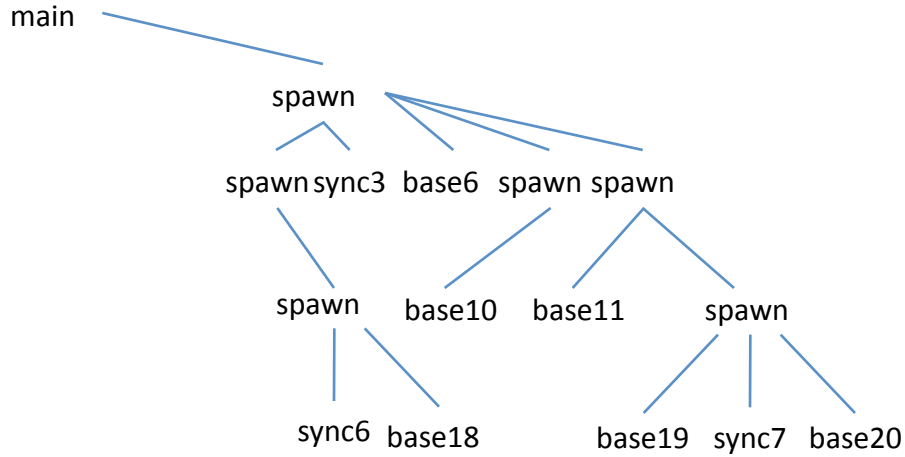
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;

Static elaboration



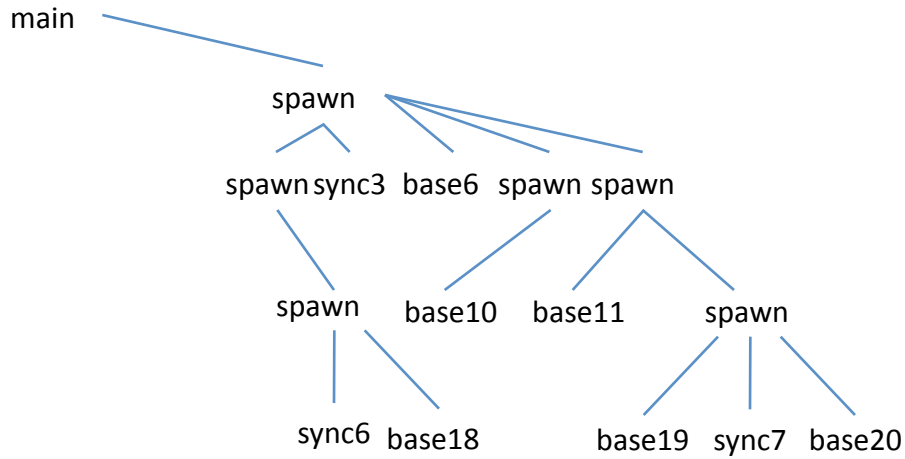
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17;

Static elaboration



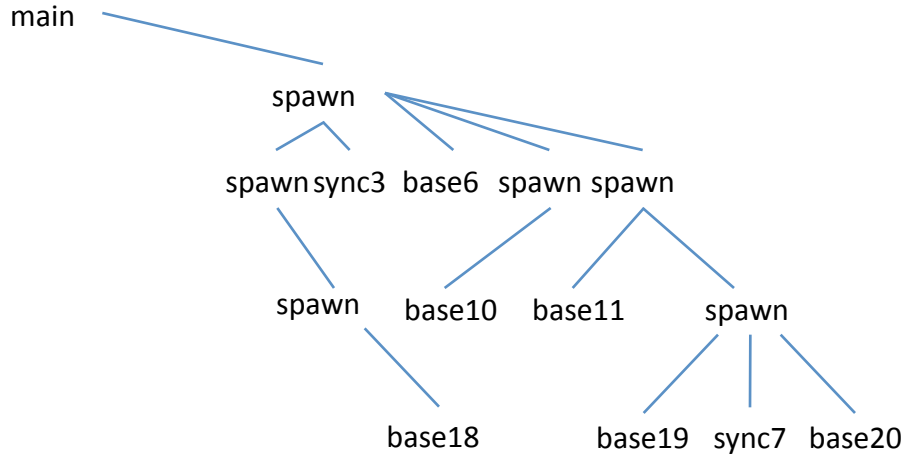
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17;

Static elaboration



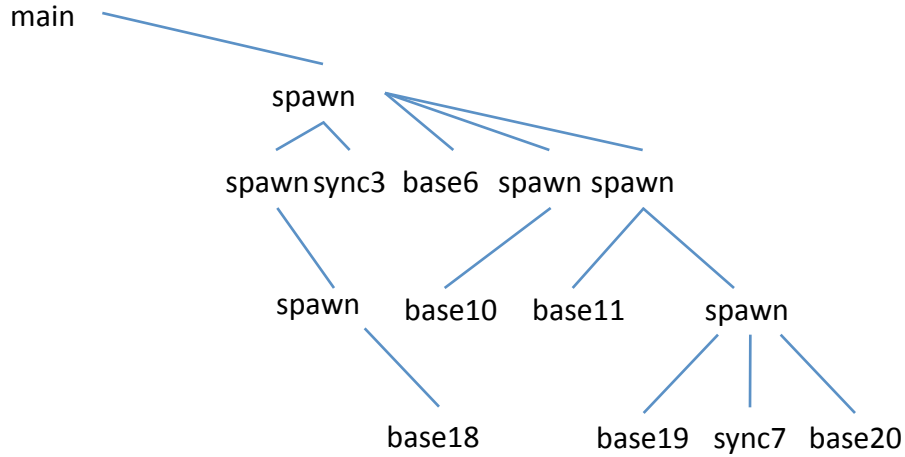
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6;

Static elaboration



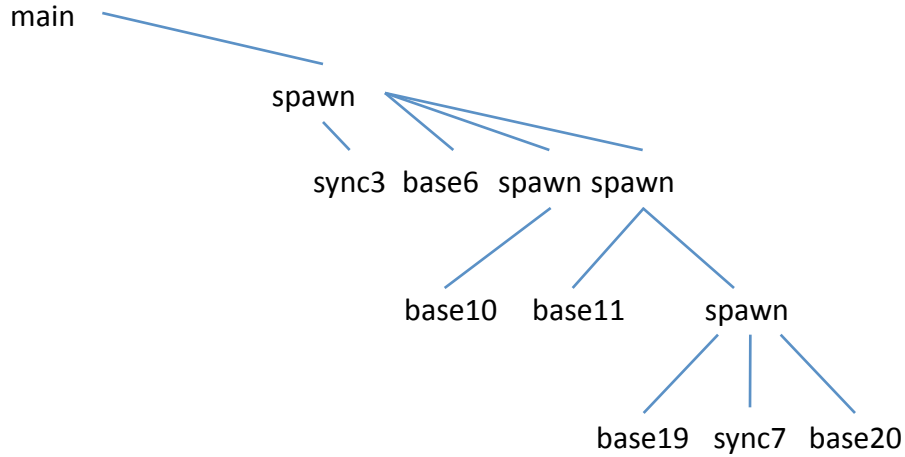
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6;

Static elaboration



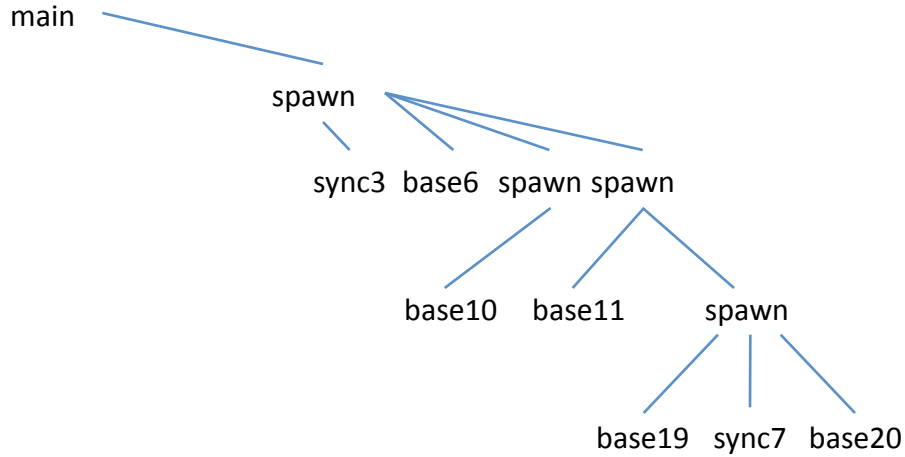
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18;

Static elaboration



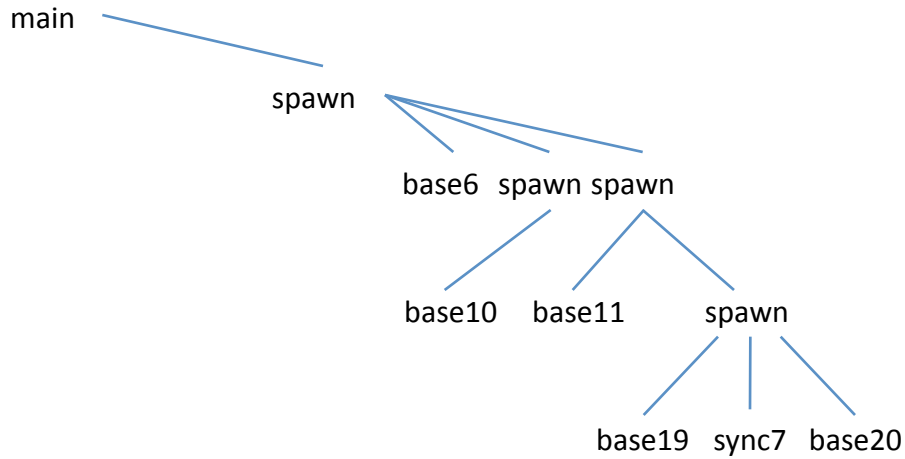
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18;

Static elaboration



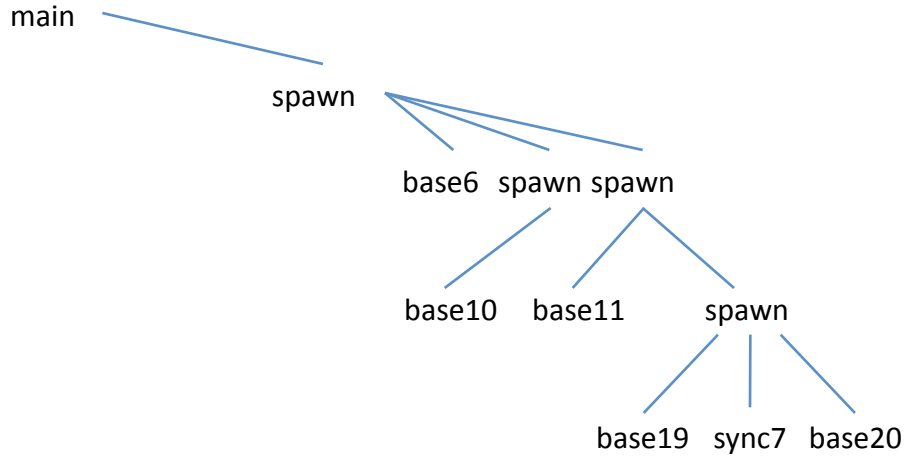
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;

Static elaboration



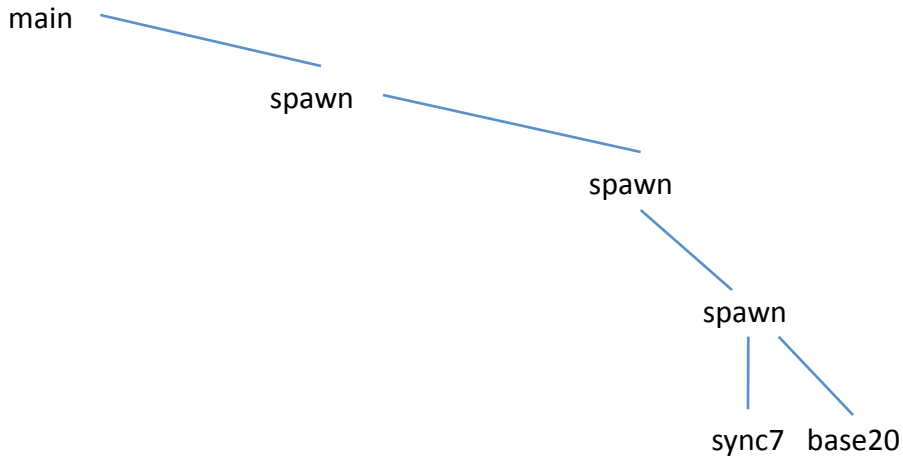
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;

Static elaboration



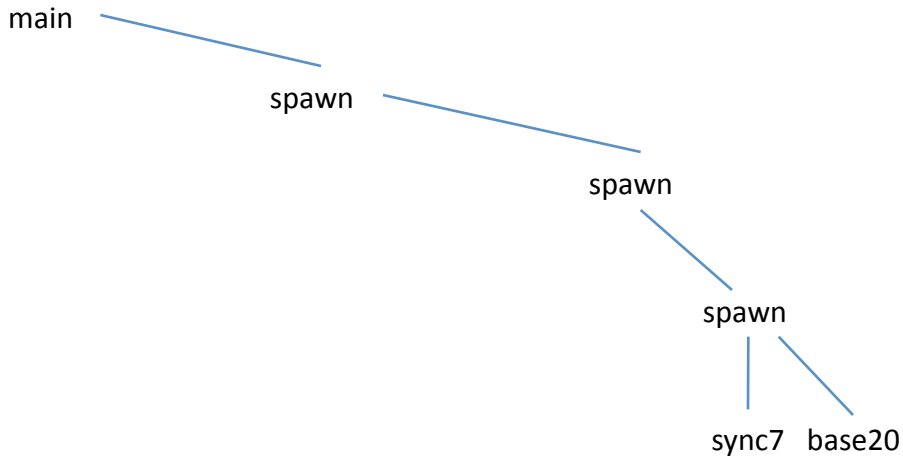
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3; spawn base6;
Spawn base10; spawn base11; spawn base19;

Static elaboration



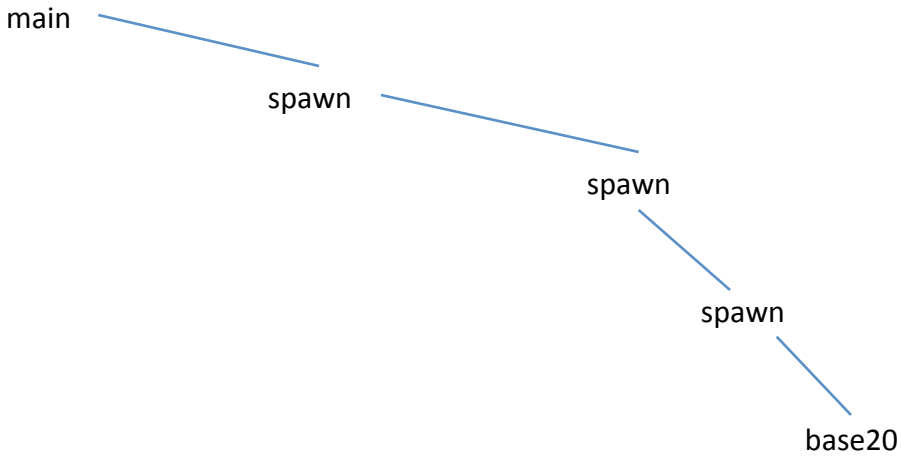
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3; spawn base6;
Spawn base10; spawn base11; spawn base19;

Static elaboration



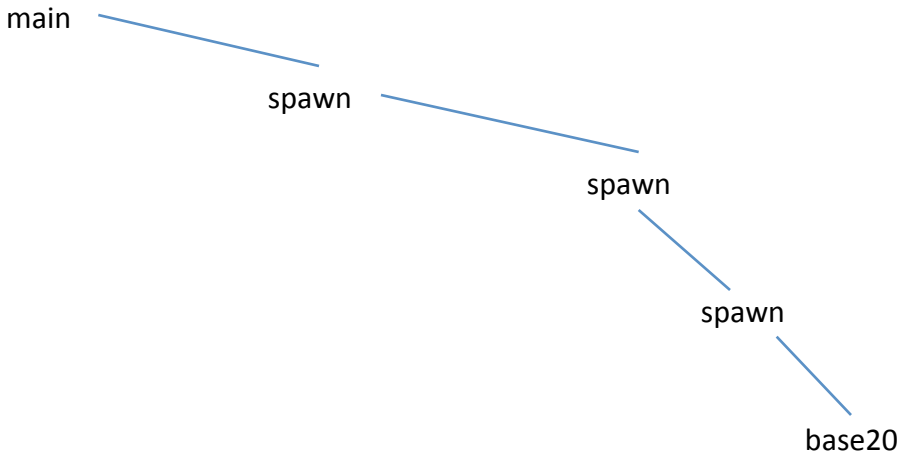
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3; spawn base6;
Spawn base10; spawn base11; spawn base19; sync7;

Static elaboration



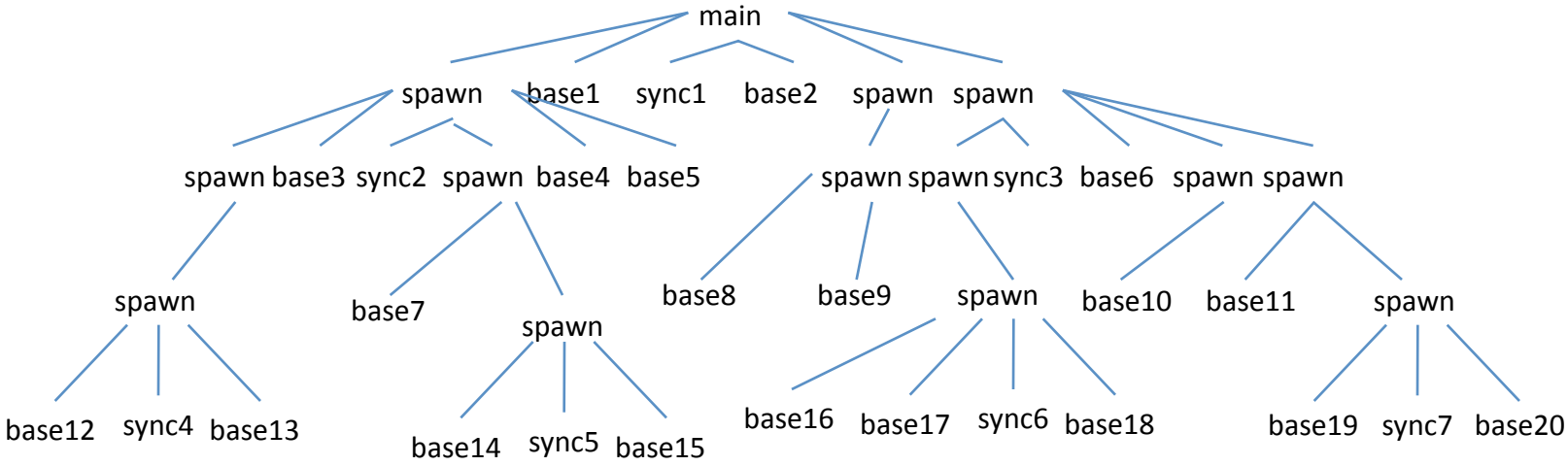
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3; spawn base6;
Spawn base10; spawn base11; spawn base19; sync7;

Static elaboration



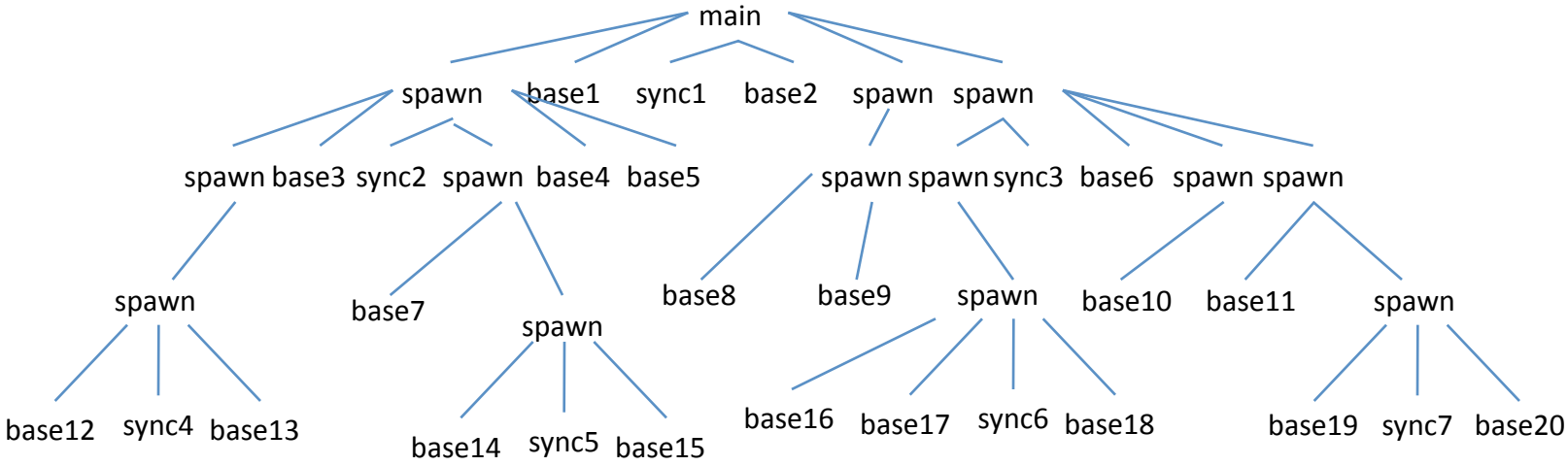
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7; spawn base14;
Spawn base4; spawn base5; sync5; spawn base15; sync1;
Spawn base2; spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3; spawn base6;
Spawn base10; spawn base11; spawn base19; sync7;
Spawn base20;

Static elaboration




```
Cilk_main stencil(){
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;
spawn base14; Spawn base4;  spawn base5;
sync5; spawn base15; sync1; Spawn base2;
spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;
spawn base6; Spawn base10; spawn base11;
spawn base19; sync7; Spawn base20;
}
```

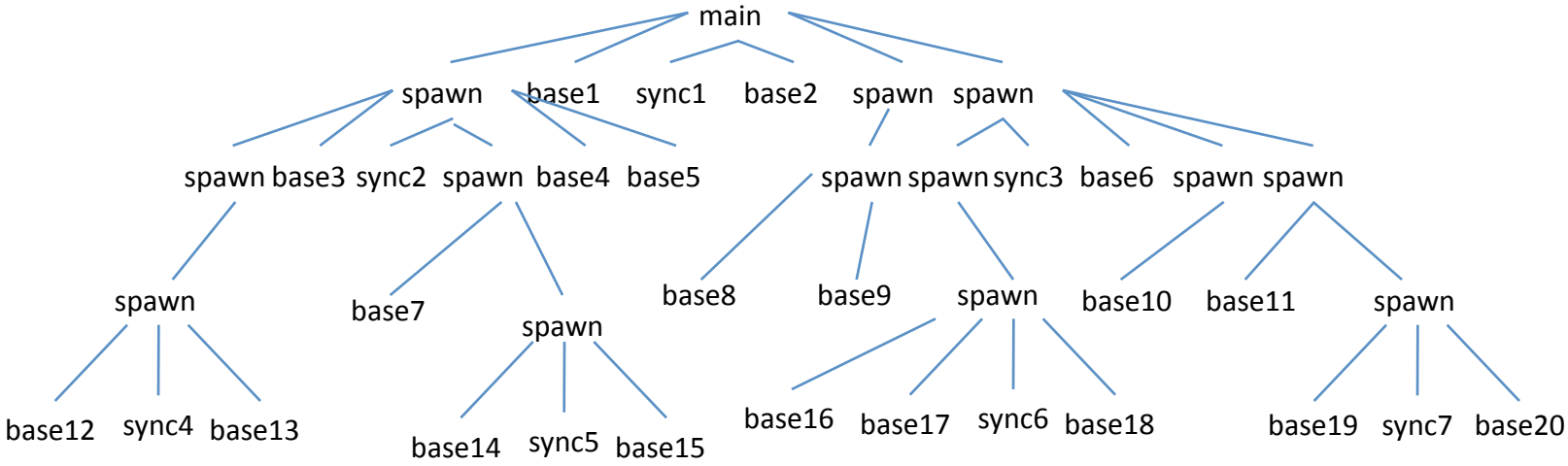
Static elaboration



Cilk_main stencil(){
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;
spawn base14; Spawn base4; spawn base5;
sync5; spawn base15; sync1; Spawn base2;
spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;
spawn base6; Spawn base10; spawn base11;
spawn base19; sync7; Spawn base20;
}

1) DFS until we meet 'sync' at **all** depth, which means we can't proceed without removing some 'sync'

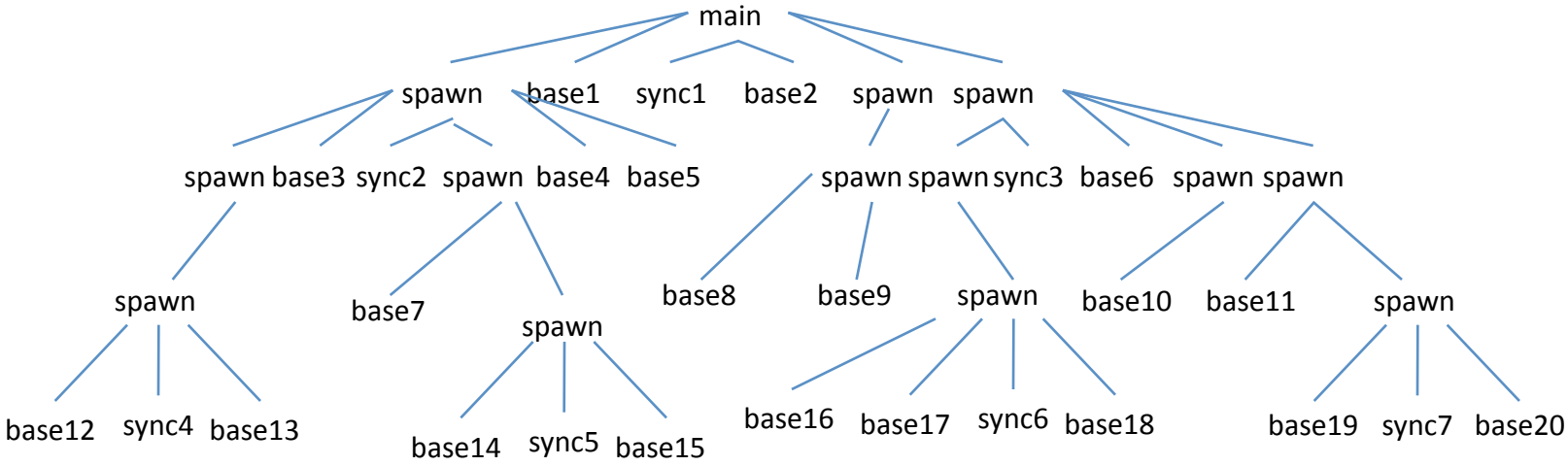
Static elaboration




```
Cilk_main stencil(){
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;
spawn base14; Spawn base4;  spawn base5;
sync5; spawn base15; sync1; Spawn base2;
spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;
spawn base6; Spawn base10; spawn base11;
spawn base19; sync7; Spawn base20;
}
```

1) DFS until we meet 'sync' at **all** depth, which means we can't proceed without removing some 'sync'
2) Remove all the visited leaves, if an internal node has no children, remove it as well.

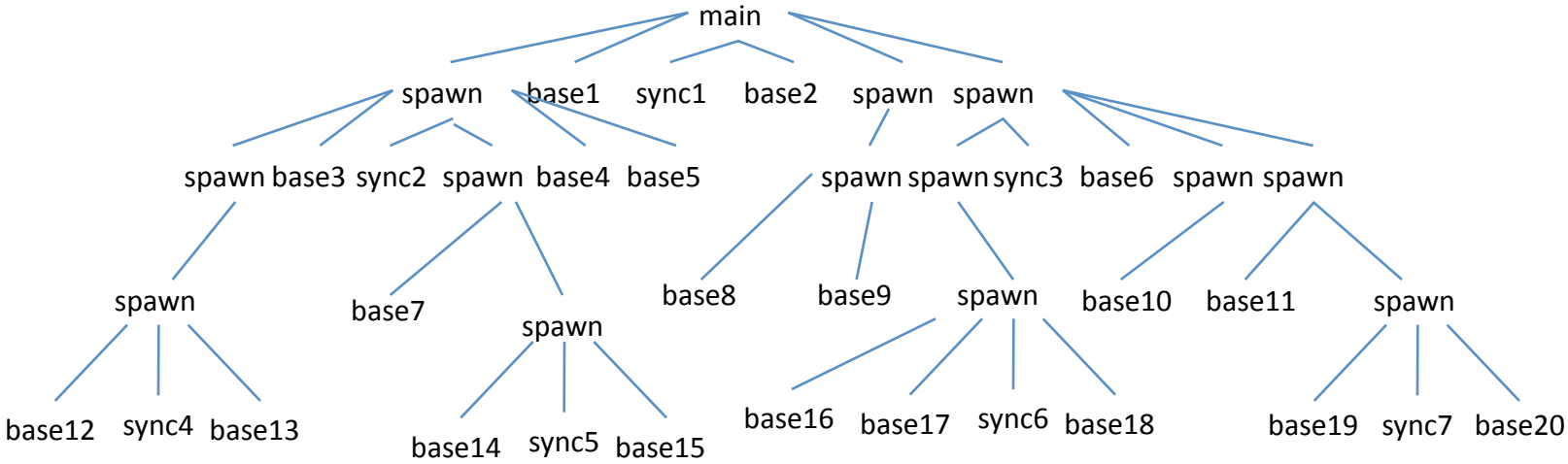
Static elaboration



Cilk_main stencil(){
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;
spawn base14; Spawn base4; spawn base5;
sync5; spawn base15; sync1; Spawn base2;
spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;
spawn base6; Spawn base10; spawn base11;
spawn base19; sync7; Spawn base20;
}

1) DFS until we meet 'sync' at **all** depth, which means we can't proceed without removing some 'sync'
2) Remove all the visited leaves, if an internal node has no children, remove it as well.
3) Remove all the 'sync's that is the biggest brother of its parent.

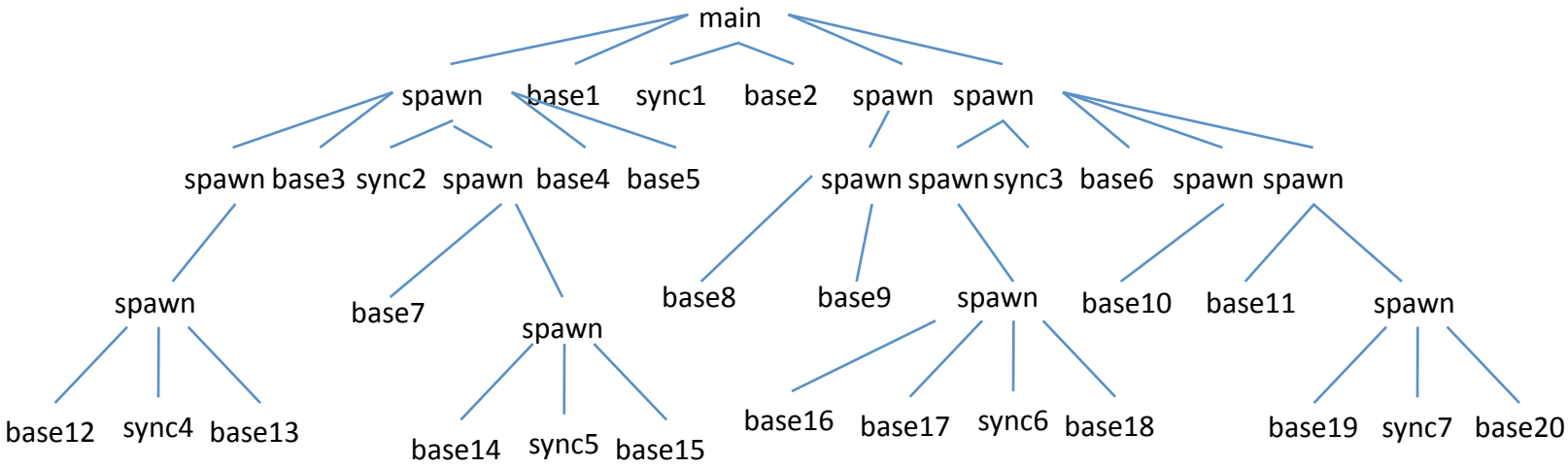
Static elaboration



Cilk_main stencil(){
Spawn base12; spawn base3; spawn base1;
Sync4; spawn base13; sync2; spawn base7;
spawn base14; Spawn base4; spawn base5;
sync5; spawn base15; sync1; Spawn base2;
spawn base8; spawn base9; spawn base16;
Spawn base17; sync6; spawn base18; sync3;
spawn base6; Spawn base10; spawn base11;
spawn base19; sync7; Spawn base20;
}

1) DFS until we meet 'sync' at **all** depth, which means we can't proceed without removing some 'sync'
2) Remove all the visited leaves, if an internal node has no children, remove it as well.
3) Remove all the 'sync's that is the biggest brother of its parent.
4) Continue DFS and visit the rest of leaf nodes from the first removed 'sync' /or from the root (because we have already removed all visited children, empty internal, proper 'sync's)

Static elaboration



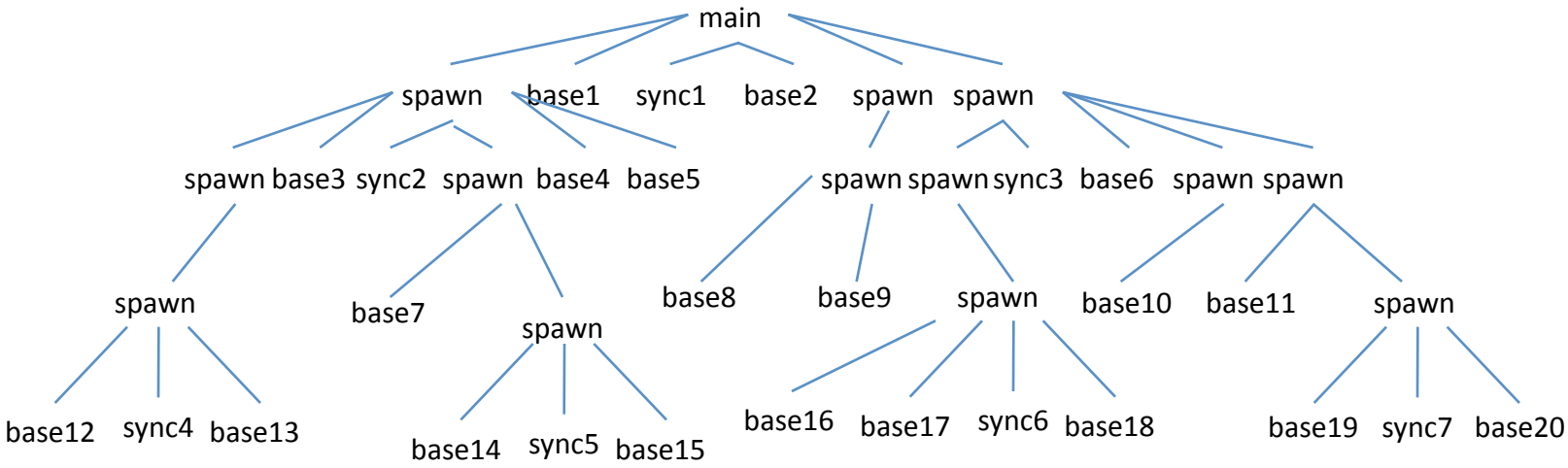
Instruction Cache:

```
Cilk_main stencil(){
   if (isRegion)
      cilk_spawn base(region i);
      //cilk_spawn walk_recursive(region i);
      //cilk_spawn walk_loop(region i);
   else if (isSync)
      cilk_sync;
   else
       return;
}
```

Data Cache:

Region1;
Region2;
Region3;
Sync;
Region4;
…….
…….
…….
Region n;
Region n+1;
…….

Static elaboration



Instruction Cache:

```
Cilk_main stencil(){
   int offset = 0;
   for (int j = 0; sync_data[j] != End; j++) {
       cilk_for (int i = offset; i < sync_data[j]; i++) {
           base_case_kernel(base_data[i]);
       }
    offset = sync_data[j];
    }
}
```

Still requires compressing region_info

Data Cache:

```
Meta_region_info
      base_data[] = {
Region1;
Region2;
Region3;
Region4;
…….
…….
…….
Region n;
Region n+1;
…….}
```

Data Cache:

```
Meta_index_info
      sync_data[] = {
index1;
index2;
index3;
index4;
…….
…….
…….
index n;
index n+1;
…….}
```

Faster : Always cut into the largest spatial dimension

- Shall we always cut into the largest dimension when comes to a space cut?

Faster : Always cut into the largest spatial dimension

- Shall we always cut into the largest dimension when comes to a space cut?
- No significant performance gains comparing with cut in order.