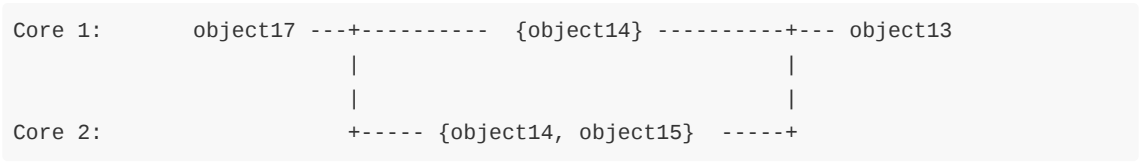
# Fork-Join Task Construction with fjlib

The fork-join library, fjlib, assigns static schedules to each core of a fork join task. The schedules are divided into parallel sections which are preceded by a fork node and followed by a join node (a fork node may also be a join node).

The library requires the user to define the fork, join, and fork-join nodes in an array named fjnodes. Task execution begins with the first function pointed to by the first element of fjnodes. After completion, the execution of the first parallel section begins. Upon completing the parallel section, the second element of fjnodes is executed before the second parallel section. This process repeats until reaching a NULL entry in the fjnodes array. By convention, core 1 is dedicated to the execution of fork and join nodes and is named fjnodes1. Any NON-NULL entries in fjnodes[2-7] are an error and will lead to undefined behavior.

For each core N there is a parallel section array: pnodesN. Each element of a pnodesN array is an array of function pointers. During the execution of a parallel section each element of the array for the section will execute in order. Thereby, each core will execute in order the functions pointed to by each element of the parallel section array. After completing a parallel section, the core will wait until signaled by the controller core 0. An example fork-join task can be found below:




In the example above there are two distinct core schedules. The first core contains one object within the parallel section flanked by a fork node (object17) and a join node (object13). The fork node, `fjnodes1[0] - object17`, is succeeded by a series of pnodes from Core 1 and Core 2 that represent the parallel section. All nodes within this parallel section share a single successor executing on Core 1, `fjnodes[1] - object14`, the join node. The process of execution can be seen as:

1. `Core 1: fjnodes[0] - object17` - executes object17 and upon completion forks pnodes1[0] and pnodes2[0]
2. Parralel Section Execution:
   2.1 - `Core 1: pnodes1[0] - {object14}` - awaits forking, upon being forked execute object14 and join at `fjnodes1[1]` upon completion.
   2.2 - `Core 2: pnodes2[0]- {object14, object15}` - awaits forking, upon being forked execute object14 until completion then object15. After both have completed join at `fjnodes1[1]`.
3. `Core 1: fjnodes1[1] - object14` - awaits for all predecessors to join and will execute object14 after.

An example of this process in code format can be found below:

```
// NUM_SECTIONS = 1
// MAX_SEC_THREADS = 256
object_t *fjnodes1[NUM_SECTIONS + 1] = {
        object17,
        object13
};
```

```
object_t *pnodes1[NUM_SECTIONS][MAX_SEC_THREADS + 1] = {
    {
        object14,
    }
};

object_t *fjnodes2[NUM_SECTIONS + 1] = {
};

object_t *pnodes2[NUM_SECTIONS][MAX_SEC_THREADS + 1] = {
    {
        object14,
        object15
    }
};
```

# Object Representations of Benchmark Tasks

As representative benchmarks, the Mälardalen Real Time Benchmarks (MRTC) have been converted to objects for fork-join task creation. The MRTC benchmarks are referred to as objects to avoid redundant linker scripts. The linker script in use is available at `fjcolo/simul/fjlib/riscv32-virt.ld`. The table below maps the MRTC benchmark name to object name.

| benchmark | object # | | benchmark | object # |
|---|---|---|---|---|
| bs | object01 | | bsort100 | object02 |
| crc | object03 | | expint | object04 |
| fft | object05 | | insertsort | object06 |
| jfdctint | object07 | | lcdnum | object08 |
| matmult | object09 | | minver | object10 |
| ns | object11 | | nsichneu | object12 |
| qurt | object13 | | select | object14 |
| simple | object15 | | sqrt | object16 |
| statemate | object17 | | ud | object18 |