Slurm select/cons_res Plugin specification

Job_test.c/_select_nodes() function to handle both core and node level details to give number of cpus allocated for every nodes. Some nodes can be allocated zero cpus.

- Core level allocation is handled in _get_res_usage function especially _allocate_cores wrapper function calling _allocate_sc
- Node level allocation is handled in _choose_nodes and _eval_nodes function.

In the below figure-1 we can get overall view of the interaction and side notes gives more details about the function and interaction. Algorithm for important functions will be explained further.

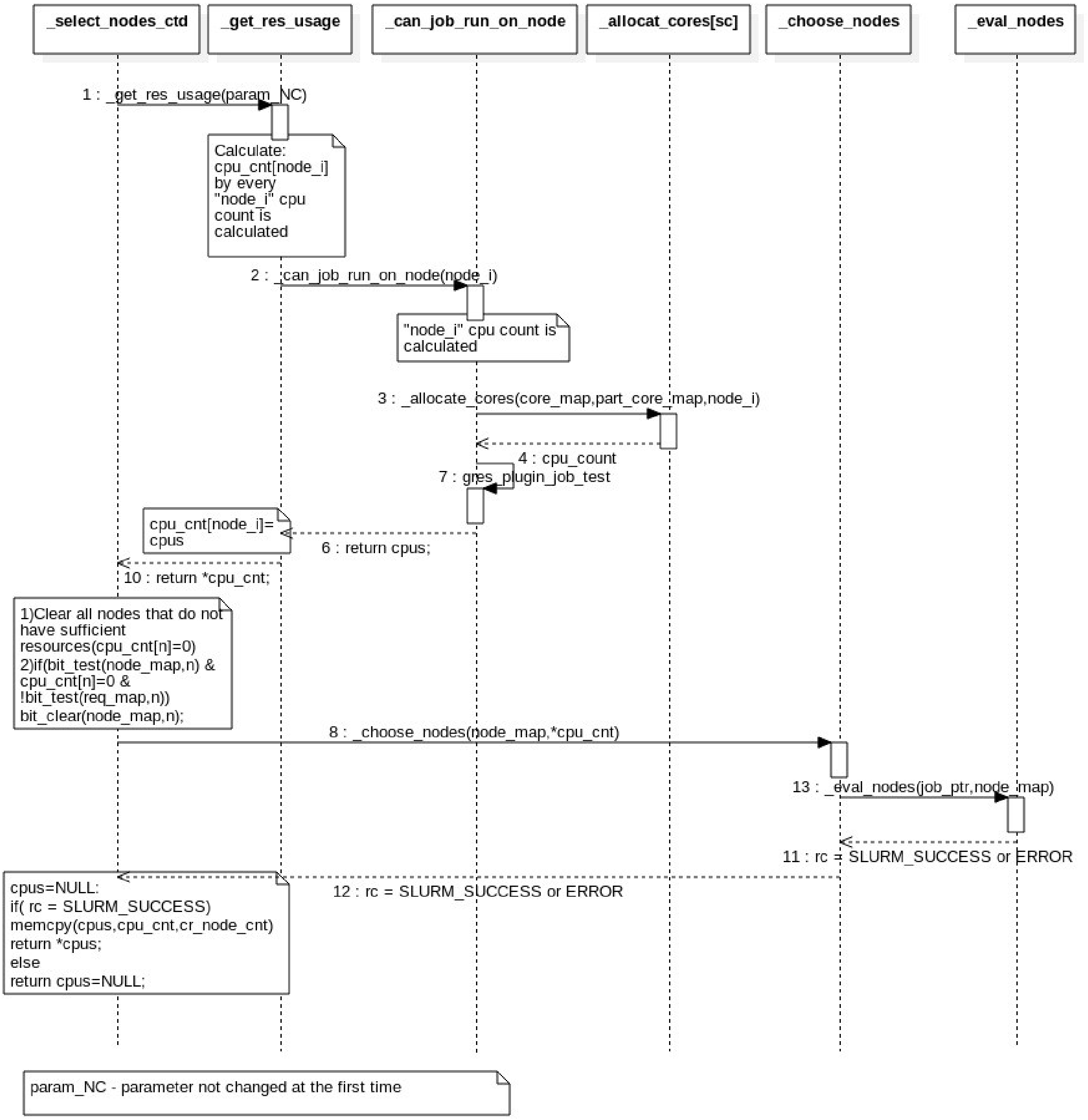
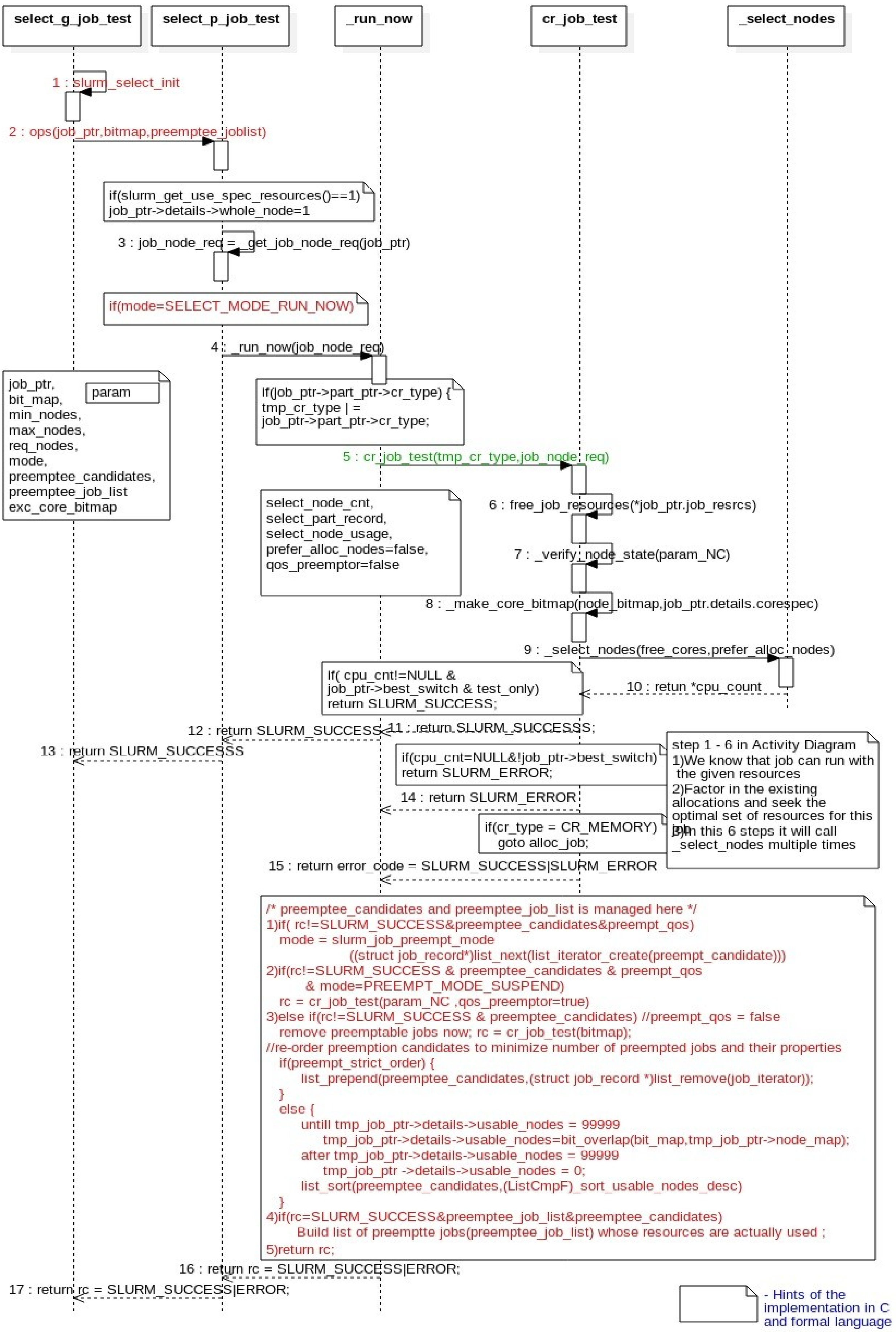


Figure-1-selectnodes function interaction details

Before entering the select_nodes it has to pass from node_select.c common function named _select_g_job_test() to gather available nodes bitmap and jobs details to select specific plugin for **selection** process as configured in slurm.conf file. In the fig-2 we can see the connection between _select_g_job_test() to _select_nodes() function and high level node manipulation also.


select_g_job_test
select_p_job_test
_run_now
cr_job_test
_select_nodes
1 : slurm_select_init
2 : ops(job_ptr,bitmap,preemptee_joblist)
3 : job_node_req = _get_job_node_req(job_ptr)
4 : _run_now(job_node_req)
param
5 : cr_job_test(tmp_cr_type,job_node_req)
6 : free_job_resources(*job_ptr.job_resrcs)
7 : _verify_node_state(param_NC)
8 : _make_core_bitmap(node_bitmap,job_ptr.details.corespec)
9 : _select_nodes(free_cores,prefer_alloc_nodes)
10 : retun *cpu_count
11 : return SLURM_SUCCESSS;
12 : return SLURM_SUCCESS
13 : return SLURM_SUCCESSS
14 : return SLURM_ERROR
15 : return error_code = SLURM_SUCCESS|SLURM_ERROR
16 : return rc = SLURM_SUCCESS|ERROR;
17 : return rc = SLURM_SUCCESS|ERROR;
- Hints of the implementation in C and formal language

```
if(slurm_get_use_spec_resources()==1)
job_ptr->details->whole_node=1
```

```
if(mode=SELECT_MODE_RUN_NOW)
```

```
job_ptr,
bit_map,
min_nodes,
max_nodes,
req_nodes,
mode,
preemptee_candidates,
preemptee_job_list
exc_core_bitmap
```

```
if(job_ptr->part_ptr->cr_type) {
tmp_cr_type | =
job_ptr->part_ptr->cr_type;
```

```
select_node_cnt,
select_part_record,
select_node_usage,
prefer_alloc_nodes=false,
qos_preemptor=false
```

```
if( cpu_cnt!=NULL &
job_ptr->best_switch & test_only)
return SLURM_SUCCESS;
```

```
if(cpu_cnt=NULL&!job_ptr->best_switch)
return SLURM_ERROR;
```

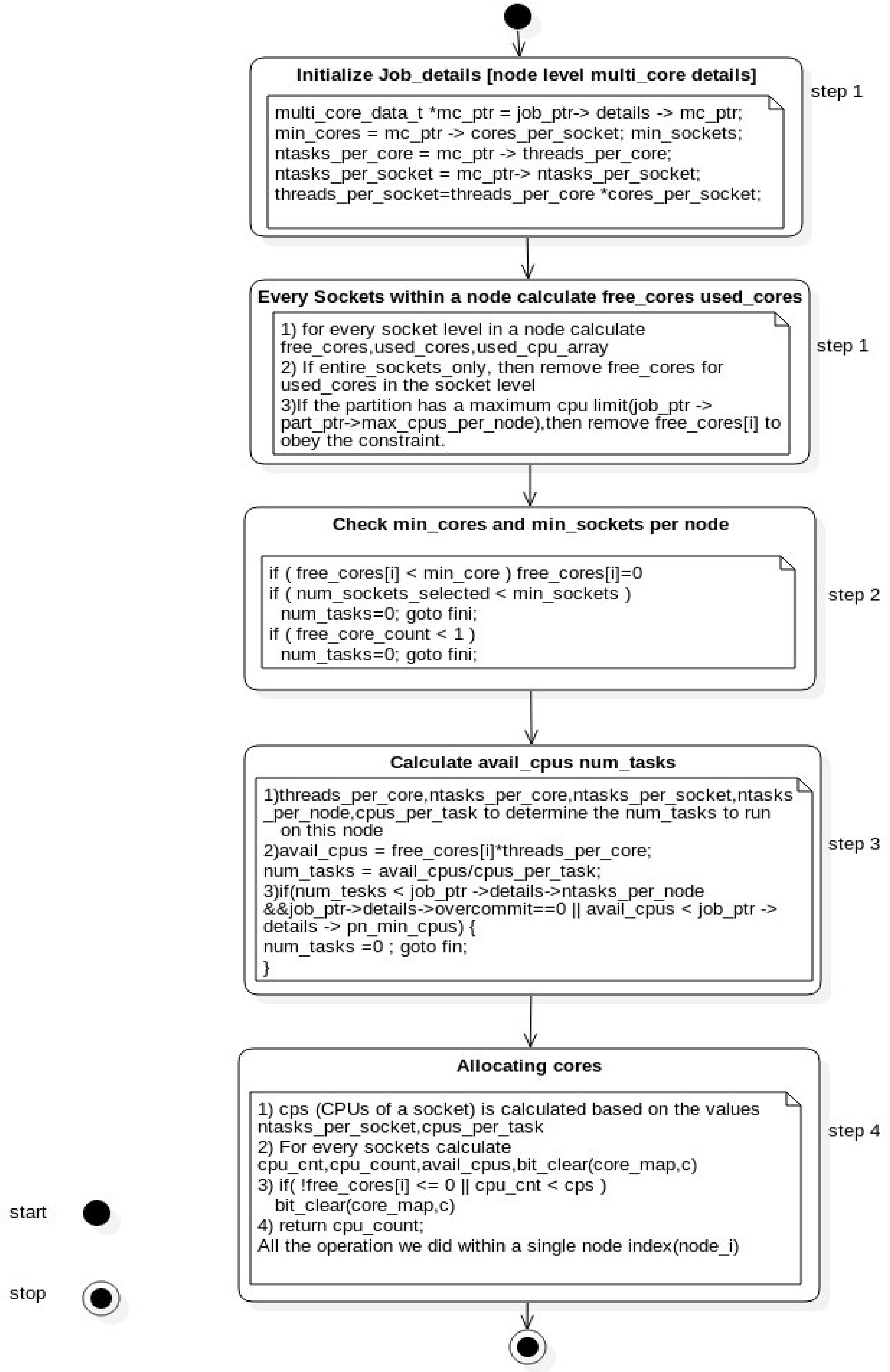
```
if(cr_type = CR_MEMORY)
  goto alloc_job;
```

step 1 - 6 in Activity Diagram
1)We know that job can run with
 the given resources
2)Factor in the existing
allocations and seek the
optimal set of resources for this
job
3)In this 6 steps it will call
_select_nodes multiple times

```
/* preemptee_candidates and preemptee_job_list is managed here */
1)if( rc!=SLURM_SUCCESS&preemptee_candidates&preempt_qos)
   mode = slurm_job_preempt_mode
                  ((struct job_record*)list_next(list_iterator_create(preempt_candidate)))
2)if(rc!=SLURM_SUCCESS & preemptee_candidates & preempt_qos
        & mode=PREEMPT_MODE_SUSPEND)
   rc = cr_job_test(param_NC ,qos_preemptor=true)
3)else if(rc!=SLURM_SUCCESS & preemptee_candidates) //preempt_qos = false
   remove preemptable jobs now; rc = cr_job_test(bitmap);
//re-order preemption candidates to minimize number of preempted jobs and their properties
   if(preempt_strict_order) {
       list_prepend(preemptee_candidates,(struct job_record *)list_remove(job_iterator));
   }
   else {
       untill tmp_job_ptr->details->usable_nodes = 99999
           tmp_job_ptr->details->usable_nodes=bit_overlap(bit_map,tmp_job_ptr->node_map);
       after tmp_job_ptr->details->usable_nodes = 99999
           tmp_job_ptr ->details->usable_nodes = 0;
       list_sort(preemptee_candidates,(ListCmpF)_sort_usable_nodes_desc)
   }
4)if(rc=SLURM_SUCCESS&preemptee_job_list&preemptee_candidates)
      Build list of preemptte jobs(preemptee_job_list) whose resources are actually used ;
5)return rc;
```

Figure-2- Entry from the select plugin to select_nodes function to handle insufficient resources in consumable manner from preempt_candidates job_list resources.

_select_nodes function will get selected node_map and core_map and to allocate specific nodes and cores(CPUs) to the job. Core level allocation for every single node is handled in allocate_core. More details about the allocate_core function is given in fig-3 below.



Initialize Job_details [node level multi_core details]
step 1

```
multi_core_data_t *mc_ptr = job_ptr-> details -> mc_ptr;
min_cores = mc_ptr -> cores_per_socket; min_sockets;
ntasks_per_core = mc_ptr -> threads_per_core;
ntasks_per_socket = mc_ptr-> ntasks_per_socket;
threads_per_socket=threads_per_core *cores_per_socket;
```

Every Sockets within a node calculate free_cores used_cores
step 1

```
1) for every socket level in a node calculate
free_cores,used_cores,used_cpu_array
2) If entire_sockets_only, then remove free_cores for
used_cores in the socket level
3)If the partition has a maximum cpu limit(job_ptr ->
part_ptr->max_cpus_per_node),then remove free_cores[i] to
obey the constraint.
```

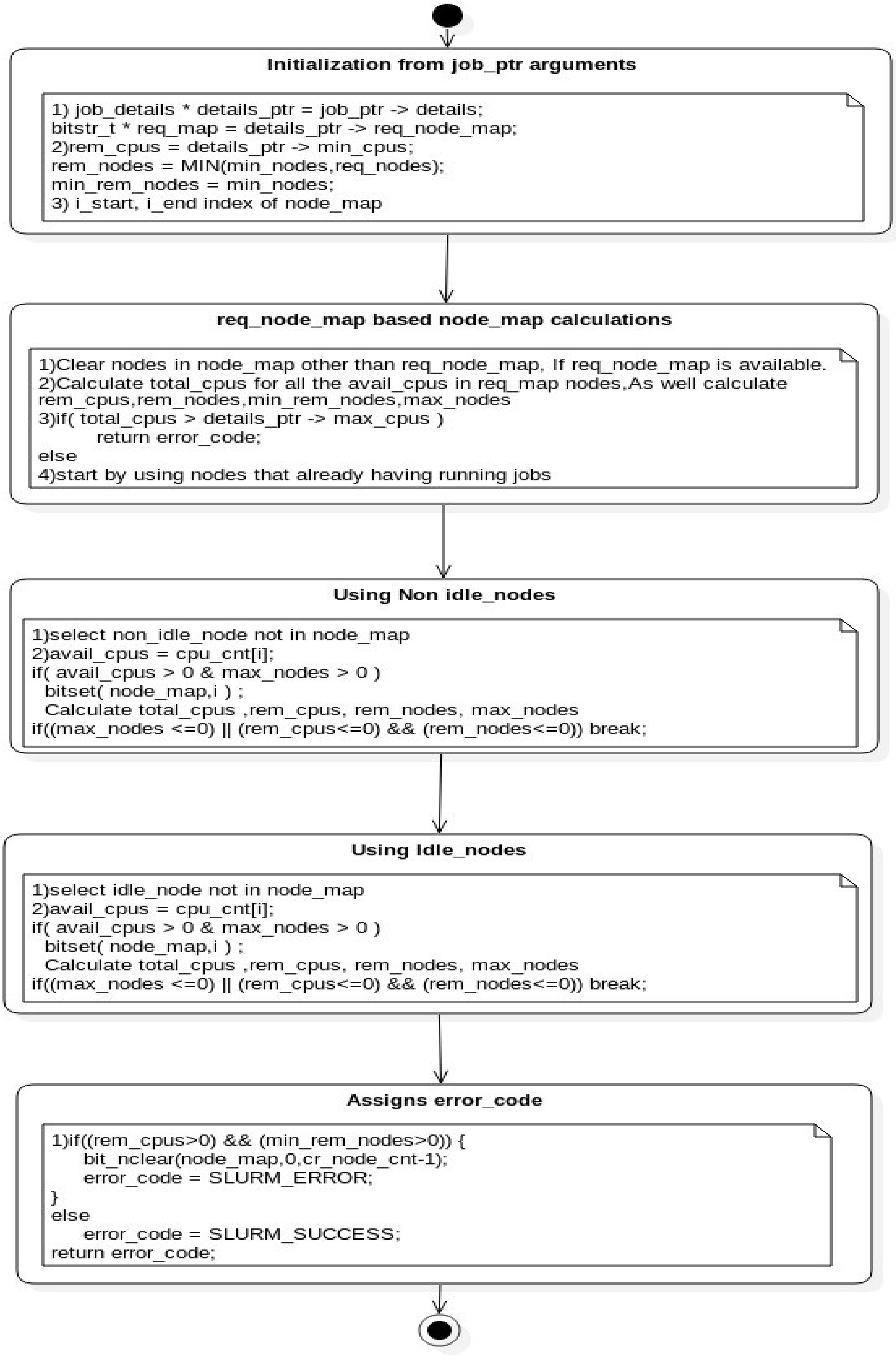
```
if ( free_cores[i] < min_core ) free_cores[i]=0
if ( num_sockets_selected < min_sockets )
  num_tasks=0; goto fini;
if ( free_core_count < 1 )
  num_tasks=0; goto fini;
```

```
1)threads_per_core,ntasks_per_core,ntasks_per_socket,ntasks
_per_node,cpus_per_task to determine the num_tasks to run
  on this node
2)avail_cpus = free_cores[i]*threads_per_core;
num_tasks = avail_cpus/cpus_per_task;
3)if(num_tesks < job_ptr ->details->ntasks_per_node
&&job_ptr->details->overcommit==0 || avail_cpus < job_ptr ->
details -> pn_min_cpus) {
num_tasks =0 ; goto fin;
}
```

```
1) cps (CPUs of a socket) is calculated based on the values
ntasks_per_socket,cpus_per_task
2) For every sockets calculate
cpu_cnt,cpu_count,avail_cpus,bit_clear(core_map,c)
3) if( !free_cores[i] <= 0 || cpu_cnt < cps )
  bit_clear(core_map,c)
4) return cpu_count;
All the operation we did within a single node index(node_i)
```

Figure-3-_allocate_sc is the wrapper function of _allocate_core. It will bit_clear cores does not match to the jobs specification and return the matched cores count for the node index(node_i).

_eval_nodes function check the nodes that matched for the jobs_specification and pick the best nodes out of the list. **_eval_nodes** function was based on the different criteria network topology, least loaded node, consecutive nodes and busy nodes only to pick the best nodes. Some of the eval_nodes criteria will be ignored by the jobs details. **_eval_nodes_busy** function select the busy nodes more preference than idle nodes from the matched nodes from the _chooed_nodes() function. **_eval_nodes_busy** function algorithm is shown below Fig-4.



Initialization from job_ptr arguments

```
1) job_details * details_ptr = job_ptr -> details;
bitstr_t * req_map = details_ptr -> req_node_map;
2)rem_cpus = details_ptr -> min_cpus;
rem_nodes = MIN(min_nodes,req_nodes);
min_rem_nodes = min_nodes;
3) i_start, i_end index of node_map
```

req_node_map based node_map calculations

```
1)Clear nodes in node_map other than req_node_map, If req_node_map is available.
2)Calculate total_cpus for all the avail_cpus in req_map nodes,As well calculate
rem_cpus,rem_nodes,min_rem_nodes,max_nodes
3)if( total_cpus > details_ptr -> max_cpus )
        return error_code;
else
4)start by using nodes that already having running jobs
```

```
1)select non_idle_node not in node_map
2)avail_cpus = cpu_cnt[i];
if( avail_cpus > 0 & max_nodes > 0 )
  bitset( node_map,i ) ;
  Calculate total_cpus ,rem_cpus, rem_nodes, max_nodes
if((max_nodes <=0) || (rem_cpus<=0) && (rem_nodes<=0)) break;
```

```
1)select idle_node not in node_map
2)avail_cpus = cpu_cnt[i];
if( avail_cpus > 0 & max_nodes > 0 )
  bitset( node_map,i ) ;
  Calculate total_cpus ,rem_cpus, rem_nodes, max_nodes
if((max_nodes <=0) || (rem_cpus<=0) && (rem_nodes<=0)) break;
```

```
1)if((rem_cpus>0) && (min_rem_nodes>0)) {
     bit_nclear(node_map,0,cr_node_cnt-1);
     error_code = SLURM_ERROR;
}
else
     error_code = SLURM_SUCCESS;
return error_code;
```