“Classical” hard & soft targets for TaMaDi

1 - Introduction: why do we go parallel?

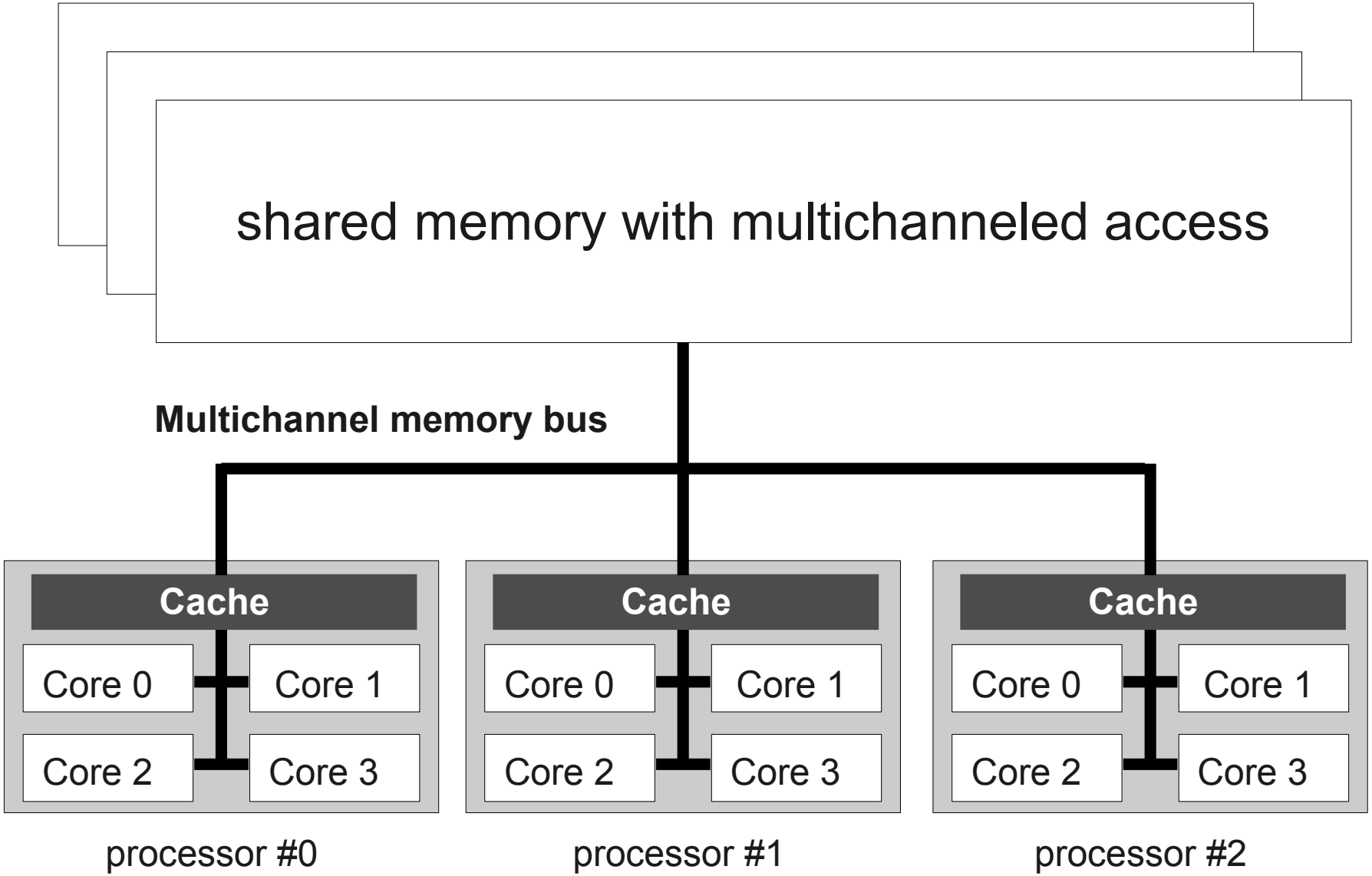
2 - Hardware targets
- **“multi/many-core” shared memory nodes;**
- **clusters of computing nodes;**
- **personnal workstations and computers.**

3 – Target parallel paradigms
- **multi-threaded applications (e.g. OpenMP);**
- **message passing parallelism (e.g. MPI);**
- **(enhanced) batch processing (e.g. BOINC);**
- **combining paradigms (slippery ground : watch your step!).**

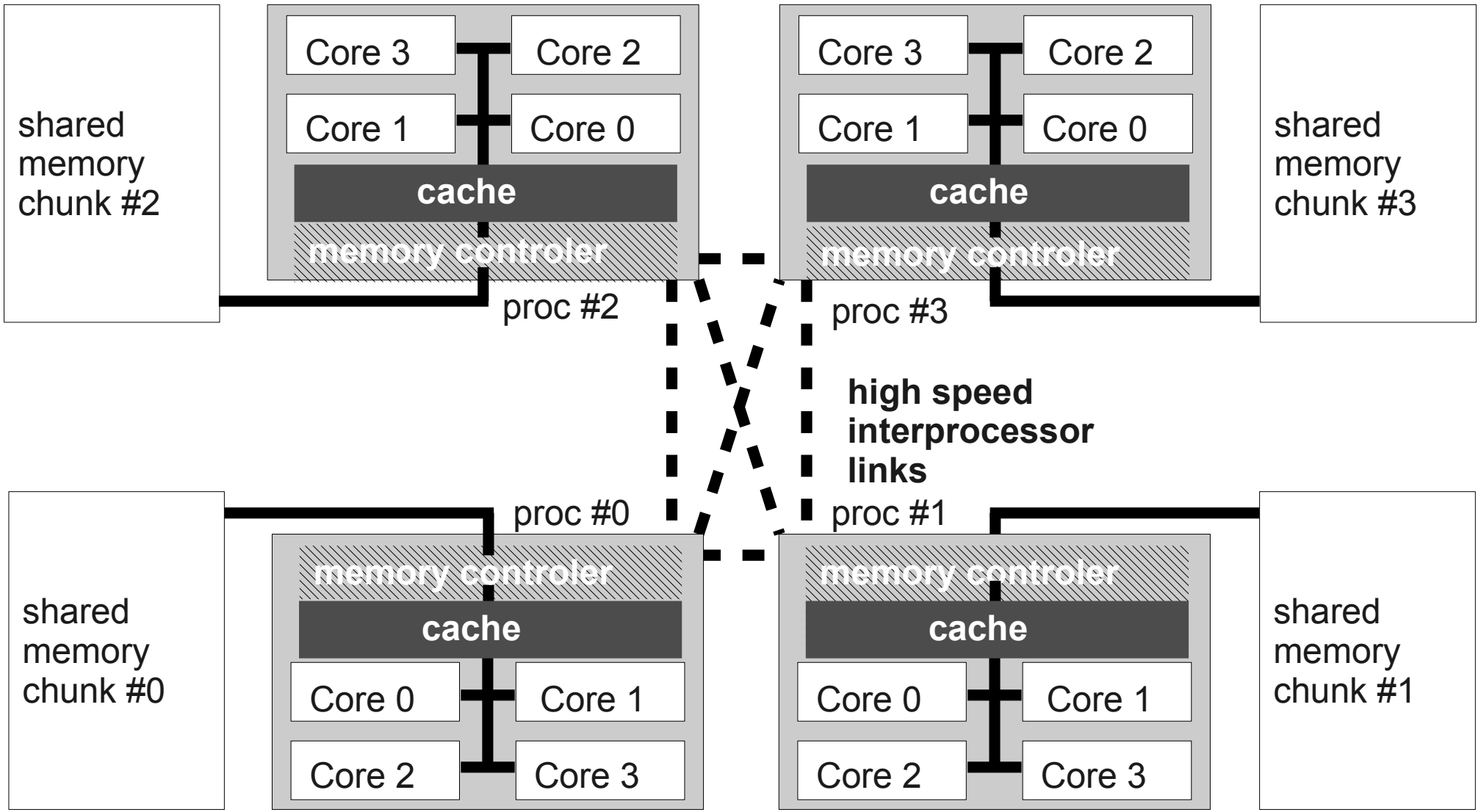
4 - A hardware platform for TaMaDi development: a proposal)

5 - Concluding remarks
- **no silver bullet for application speedup:**
  - **Amdahl’s law,**
  - **intricacies of parallel development,**
  - **retrofitting parallelism into applications is tedious and inefficient;**
- **the first step: an *efficient* serial version *with parallelisation in mind*.**

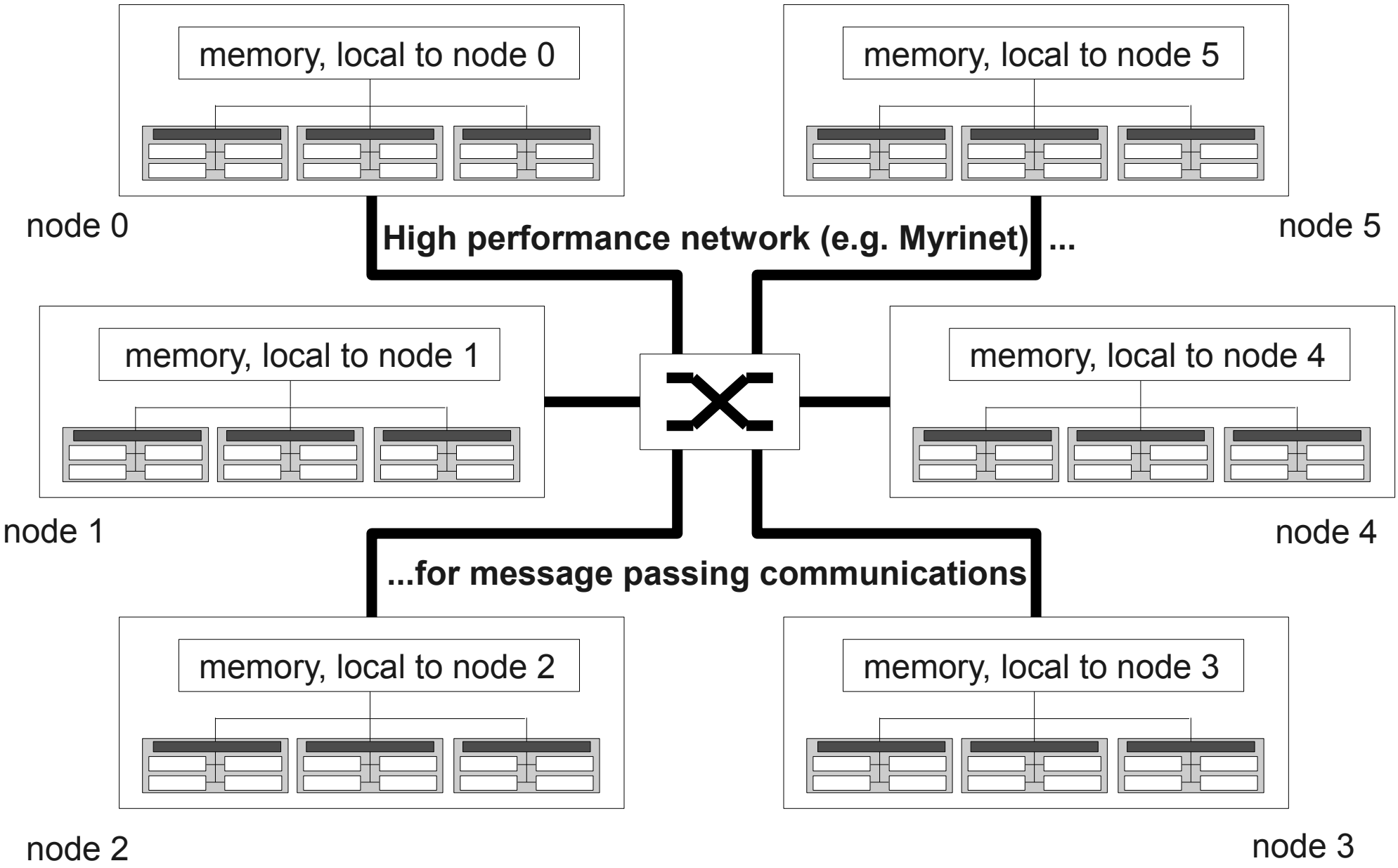
“multi/many-core” shared memory nodes 1/2



“multi/many-core” shared memory nodes 2/2

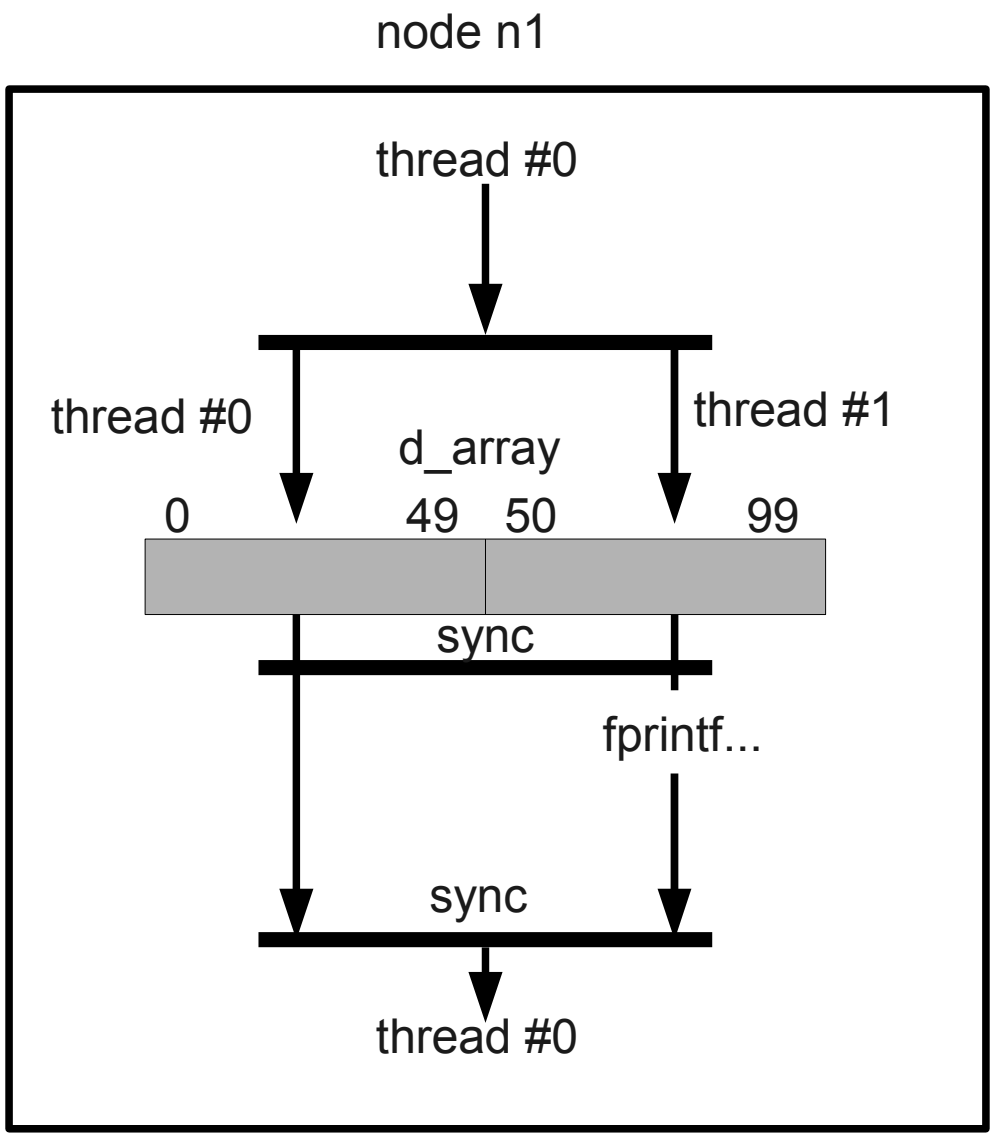


Cluster of computing nodes with local memory only



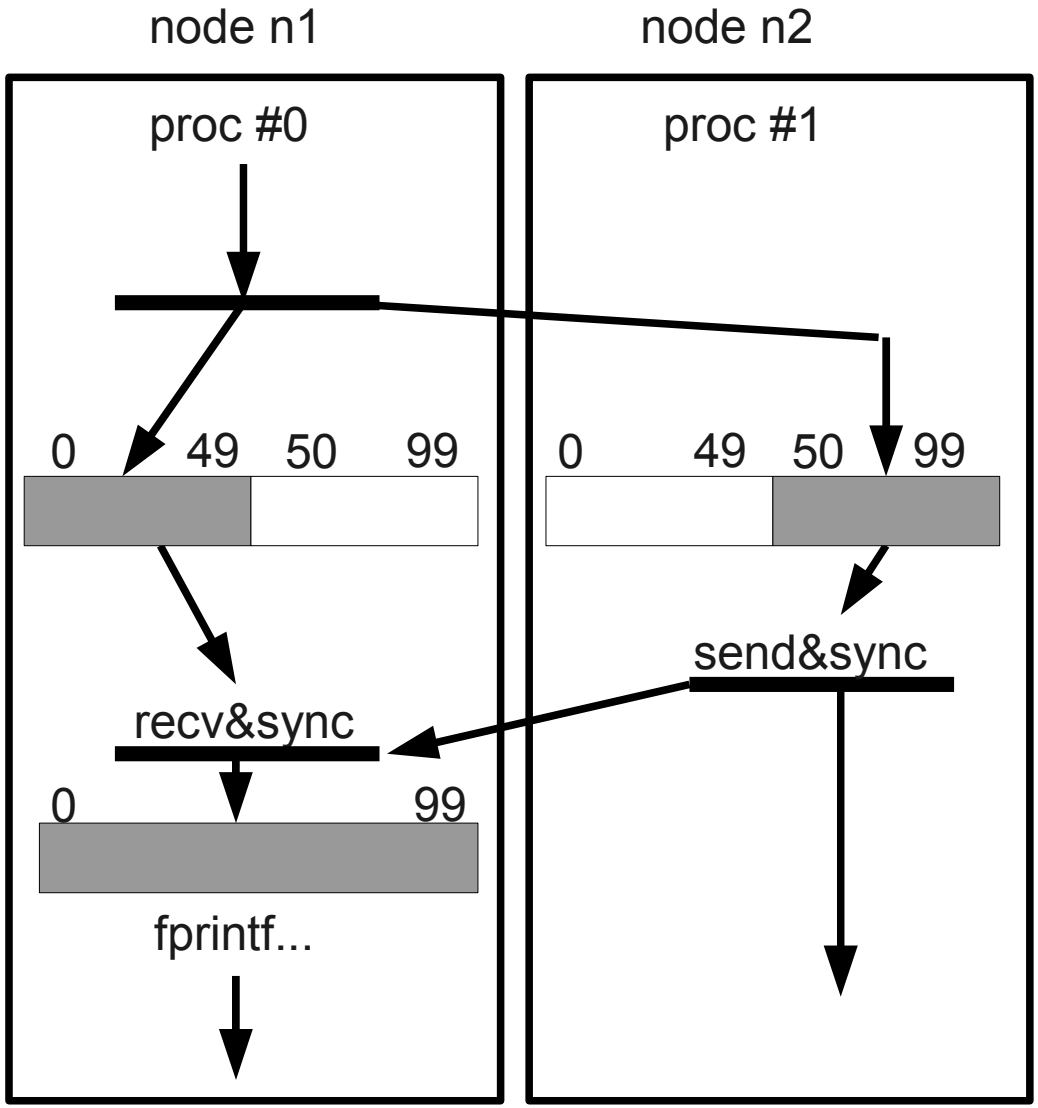
Multithreading with OpenMP

```
#include <stdio.h>
#include <omp.h>

#define ARRAY_SIZE 100

int main(int argc, char** argv)
{
  double d_array[ARRAY_SIZE];
  int i,j;

  #pragma omp parallel num_threads(2)
  {
    #pragma omp for

    for (i = 0 ; i < ARRAY_SIZE ; i++)
    {
       d_array[i] = hard_to_compute_func(i);
    } // Implicit barrier (and memory flush).

    #pragma omp single // Only on thread prints
    {
      for(j = 0 ; j < ARRAY_SIZE ; j++)
         printf("val %d: %d ", j, d_array[j]);
    }
  } // End parallel region.
  return(0);
} // End main.
```



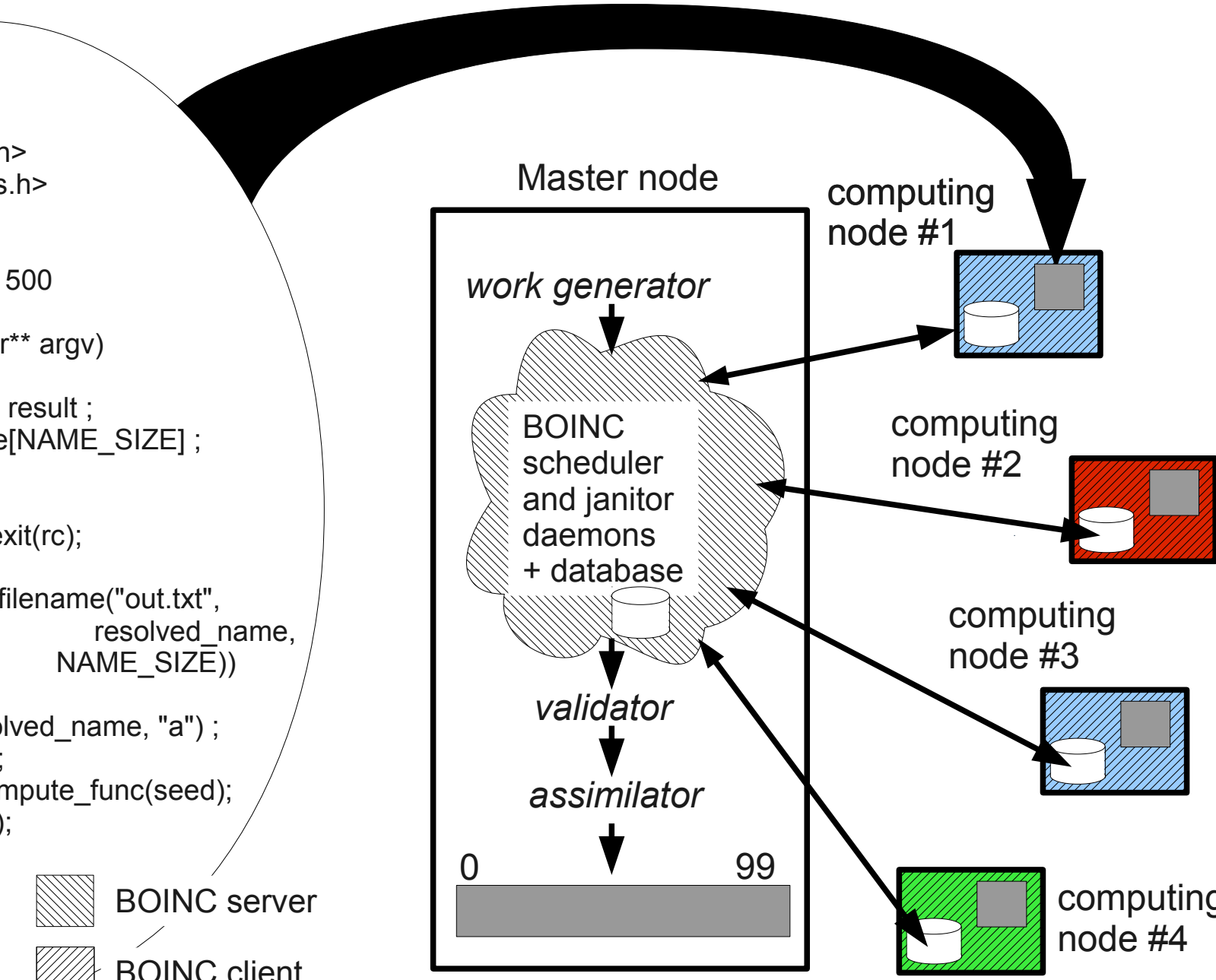
Message passing with MPI

```
#include <stdio.h>
#include <mpi.h>

#define ARRAY_SIZE 100

int main(int argc, char** argv)
{
  double d_array[ARRAY_SIZE];
  int i,j;
  int rank, size = ARRAY_SIZE / 2;
  MPI_status status;

  MPI_Init();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for (i = size * rank ; i < size * (rank+1) ; i++)
  {
    d_array[i] = hard_to_compute_func(i);
  }
 if (rank == 0)
 {
   MPI_Recv(d_array + size, size, MPI_DOUBLE,
                  1, 1, MPI_COMM_WORLD, &status);
   for(j = 0 ; j < ARRAY_SIZE ; j++)
     printf("val %d: %d ");
 }
 else
   MPI_Send(d_array + size, size, MPI_DOUBLE,
                  0, 1, MPI_COMM_WORLD)
  return(0);
} // End main.
```



Batch processing with BOINC

```
#include <stdio.h>
#include <boinc_api.h>
#include <diagnostics.h>
#include <filesys.h>
...
#define NAME_SIZE 500

int main(int argc, char** argv)
{
  int seed, rc ; double result ;
  char resolved_name[NAME_SIZE] ;
  FILE* f ;

  if(rc = boinc_init()) exit(rc);
  if(rc =
        boinc_resolve_filename("out.txt",
                                resolved_name,
                              NAME_SIZE))
      exit(rc) ;
  f = boinc_open(resolved_name, "a") ;
  seed = atoi(argv[1]);
  result = hard_to_compute_func(seed);
  fprintf(f, "%f", result);
  fclose(f);
  boinc_finish(0);
} // End main.
```



Amdahl's Law

$$S = \frac{1}{(1-P) + \frac{P}{S_p}}$$

S : global speedup
P : parallel portion.
S_p : speed up over the parallel portion.