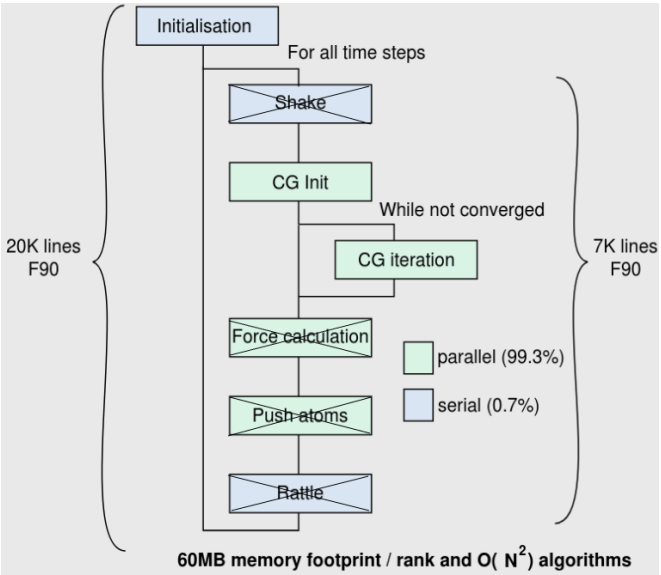
EXA2PRO SkePU and StarPU tools applied to a supercapacitors simulator (Metalwalls)

A. Description of Metalwalls:

Metalwalls is a classical molecular dynamic code created by P. Madden, currently improved and maintained by M. Salanne. The code specificity is to simulate accurately electrochemical systems like supercapacitors. Electrostatic interactions are of primary importance in this context which is why an ewald summation is used to compute it in order to maximize accuracy. The simulation computes charge density on electrodes with constant potential on them thanks to a conjugate gradient. The code is written in Fortran 90 and is parallelised with MPI. The application we are using for EXA2PRO is the core of Metalwalls stripped from its physics as can be seen on the diagram below



B. Applying the EXA2PRO framework to MetalWalls (SkePU & StarPU tools)

SkePU

In order to use SkePU, a skeleton programming framework developed by LIU, we had to port the computing kernels of our Fortran code in C++. There are three kernels in total and the simplest one at its core consists in two loops where each interaction between atoms and ions is computed. This results in an algorithm with a complexity $O(N^2)$ and a memory impact of O(N), with N the number of atoms/ions here. The implementation of SkePU in this kernel is representative of the implementations in the other two kernels, thus we will focus on this one.

SkePU was implemented in our application using the skepu::MapPairsReduce skeleton which directly "map" and "reduce" without using a costly intermediate full NxN matrix. In our code we had to refactor the computation kernels in [skepu::userfunction] which would be used by the skeleton to produce the desired output after specifying the reduction mode (row-wise or column-wise) as can be seen on the pseudo code example illustrating our implementation below.

Original C++ code:

```
double V[num_atoms], z[num_atoms], q[num_atoms];
for (i = 0; i < num_atoms; i++) {
```

```
        vi = 0.0;
        for (int j = 0; j < i; j++) {
                zij = z[j] - z[i];
                zijsq = zij*zij;
                pot_ij = exp(-zijsq) + zij*erf(zij));
                V[j] = V[j] - q[i] * pot_ij;
                vi = vi + q[j] * pot_ij;
        }
        V[i] = V[i] - vi;
}
```

SkePU-ized code:

```
[[skepu::userfunction]]
real_t map_function( skepu::Index2D i, const real_t zi, const real_t zj, skepu::Vec<real_t> q){
real_t zij = zi - zj;
real_t qj = q[i.col];
return - qj * ( exp(-zij*zij) + zij*erf(zij) );
}

[[skepu::userfunction]]
real_t plus (real_t a, real_t b) { return a + b; }

auto pairs2reduce = skepu::MapPairsReduce(map_function, plus);
pairs2reduce.setReduceMode(skepu::ReduceMode::RowWise);
pairs2reduce(V,z, z, q);
```

This refactoring increases the readability of the code making it easier to maintain and for new users to change, as they would focus on smaller parts of the whole code.

StarPU

The starting point of the StarPU implementation is bit different compared to SkePU. The implementation above treats atom pairs one by one whereas a block algorithm has been used for the StarPU implementation.

```
for (int b = 0; b < num_blocks; b++)
   for (int i = start1[b]; i < end1[b]; i++)
      for (int j = start2[b]; j < end2[j]; j++) {
         //Perform calculation of atom pair interaction on this block
      }
```

The StarPU implementation consists then to submit a task for each block instead of doing the calculation inside this loop. In the original MPI implementation, the construction of the start* and end* arrays on each rank is complex as the load has to be distributed. In the StarPU context, this part is much simpler as the task submission is a serial process, the load balancing is performed dynamically at run time.

C. Results

All tests have been run on juawei-x12 on one Node made of two CPU with ten cores each.

SkePU

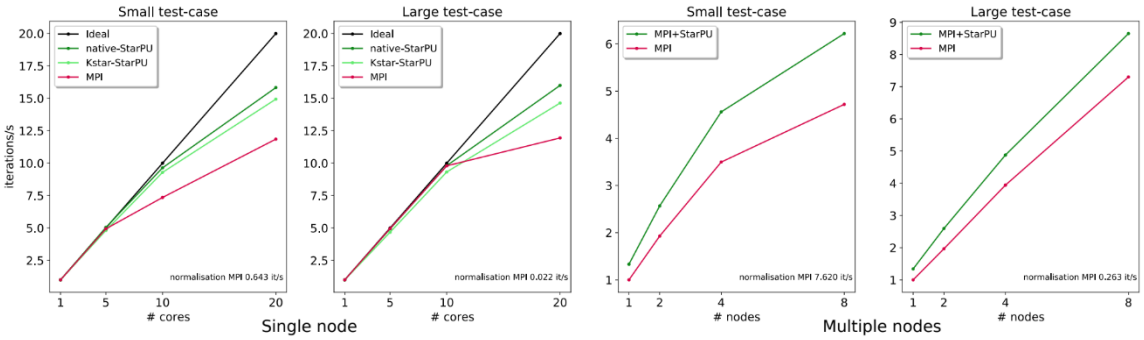
Three test cases have been used to estimate the performance of the implementation with SkePU using OpenMP backend, compared to the original heavily optimized code wich uses MPI.

		baseline	SkePU 1 core	SkePU 20 cores	OpenMPI 20 cores
Small test case	Time (s)	137,5	2235	190	7,3
	Speedup	-	x16,2 slowdown	x1,38 slowdown	x18,8 speedup
Medium test case	Time (s)	422,5	1265	104	24,2
	Speedup	-	x3 slowdown	x4,06 speedup	17,5x speedup
Big test case	Time (s)	15552	x	3048	814
	Speedup	-	x	x5,1 speedup	19,1x speedup

As can be seen on the table of the performance results below, one can notice a pretty good intra node scalability of the original application. The parallelisation leverages the small memory footprint putting a low pressure on the memory bus and thus enabling a scalability up to a full node. SkePU implementation is behind but the larger the test case, the closer the results between the SkePU implementation and our original code were. We do notice an impressive drop in performance for the smallest test case which is using mostly one the three kernels. This drop has two explanations, firstly the refactoring loses some optimization done to the algorithm itself, secondly the compiler may not be able to vectorize the SkePU-ized code as well.

StarPU

The parallelism grain can be adjusted by setting the appropriate block size. In our case, blocks of 128x128 atoms were the good compromise between small enough blocks to express parallelism and large enough blocks to leave the scheduling overhead small.



The single-node and multi-node performance of StarPU tasks in comparison to the original MPI implementation using a performance metric: number of CG iteration per second. As can be seen, the StarPU tasks clearly outperform the original MPI implementation in all cases with an increase in performance going from 18% to 33%. However, its performance drops noticeably while going from a single socket to the complete node. This could be attributed to memory affinity issues due to NUMA effects between the two sockets. Considering the multi-node scalability, one can see that the small test case already reaches its scalability limit due to its small size, while the large test case continues to scale almost perfectly.

D. Conclusions

Although the use of the SkePU framework has a performance cost, we believe this cost can be compensated in development time, maintainability and the flexibiliy to use any hardware (CPU, GPU or even FPGA) without changing the code base. StarPU works at a lower abstraction level and is thus more complex to use than SkePU. However, it eases the expression of parallelism and provides better result than the original pure MPI implementation.

More Information: G. Tzanos, V. Soni, C. Prouveur, M. Haefele, S. Zouzoula, L. Papadopoulos, S. Thibault, N. Vandenbergen, D. Pleiter, D. Soudris, “Applying StarPU Runtime System to Scientific Applications: Experiences and Lessons Learned”, POMCO 2021.