SPAWN: An Iterative, Potentials-Based, Dynamic Scheduling and Partitioning Tool

Jean-Charles Papin, Christophe Denoual, Laurent Colombet, Raymond Namyst

SPAWN: An Iterative, Potentials-Based, Dynamic Scheduling and Partitioning Tool

Jean-Charles PAPIN[1,2], Christophe DENOUAL[2], Laurent COLOMBET[2], and Raymond NAMYST[3]

[1] *CMLA, ENS-Cachan, 61 avenue du Président Wilson 94235 Cachan, FRANCE*
[2] *CEA, DAM, DIF, F-91297 Arpajon, FRANCE*
[3] *Université de Bordeaux, 351 cours de la Libération, 33400 Talence, FRANCE*

ABSTRACT

Many applications of physics modeling use regular meshes on which computations of highly variable cost can occur. Distributing the underlying cells over manycore architectures is a critical load balancing step that should increase the period until another step is required. Graph partitioning tools are known to be very effective for such problems, but they exhibit scalability problems as the number of cores and the number of cells increases. We introduce a dynamic task scheduling approach inspired by physical particles interactions. Our method allows cores to virtually move over a 2D/3D mesh of tasks and uses a Voronoi domain decomposition to balance workload among cores. Displacements of cores are the result of force computations using a carefully chosen pair potential. We evaluate our method against graph partitioning tools and existing task schedulers with a representative physical application, and demonstrate the relevance of our approach.

Keywords:. Simulation, dynamic load-balancing, tasks, manycore, pair potential, graph partitioning.

1. INTRODUCTION

Many physics simulation applications rely on large meshes where each cell contains a few elementary computing elements (e.g. particles, finite elements, or finite difference cells) and is linked to its neighboring cells. Figure 1 illustrates the concept of a *regular* mesh. The computing cost of each cell varies with the modeled material and the applied action (e.g. shock wave or distortion). Thus, distributing cells among computing units must preserve locality of neighbors and balance the computing load. Mesh partitioning can either be achieved with common task scheduling tools or graph partitioning tools.

Task-based parallelism is now a well-known concept. Along with specific libraries [2, 15, 21] and languages extensions [16], many languages now integrate tasks support in their standard [10], such as C++11 with its *future* variables concept. This allows programmers to easily exploit multiple computing units. Tasks are generally associated with a data set and can *exchange* data with other tasks, allowing to define an *affinity* criteria for a given task. Therefore, a particular attention shall be paid to the place (i.e. the computing unit) where the task will be executed. However, the still increasing cores-on-chip and chips-in-system number [11], forces applications to spawn a large number of tasks so as to prevent cores from reaching a *starving state*. This eventually leads to an important complexity: since manifold *scheduling policies* try to distribute tasks among available cores by satisfying these affinity criteria while minimising the overall application computing time, the number of combinations increases with the number of available cores and tasks.

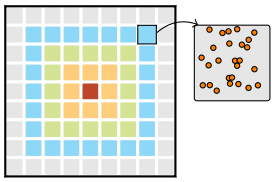

Figure 1: 2D Mesh example. The red cell has 8 direct neighbors, 16 2^{nd}*rank* neighbors and 32 3^{rd}*rank* neighbors. Every cell contains a set of elementary elements (atoms, finite elements or finite difference cells).

Exploiting the fact that tasks have strong affinities with their neighborhood (direct neighbors and neighbors of 2nd-3rd rank, see fig. 1), such meshes can direcly be represented by graphs where each node is a task, and where a link is a neighboring connection. Graph partitioning tools [18, 13] can thus be used to distribute tasks over cores in this context. Partitioning tools provide very efficient algorithms, for partitioning very large regular and non-regular graphs, which can take into account affinity characteristics. However, such tools do not perform well when dynamic scheduling is needed. Slight task weight variations will usually lead to the generation of a completely different graph partition, which will incur numerous data transfers to re-assign tasks. Support for dynamic scheduling was thus added to recent and distributed versions of partitioning libraries [8, 12, 9]: they offer *refinement* functionality by using the previous computed distribution to compute the next one. On

modern architecture (i.e. the Intel® Xeon Phi), the number of cores is significant (typically 61 cores for Intel® Xeon Phi 7100 series) and the number of available threads unit doubles or quadruples this number. The amount of memory per core is consequently a limitating factor (less than 300 Megabytes which must be shared among threads) and low memory footprint load balancing mechanisms mush thus be designed.

We present `Spawn`, a physical interaction inspired scheduler that produces compact and optimal voronoi domains. In our case, Voronoi diagrams maximize per-core data locality, by providing numerous advantages: cache usage improvements, more efficient `NUMA`-aware memory allocation/accesses and less *Point-to-Point* communications. This scheduler has the advantage of being efficient to compute and offers an automatic refinement. This paper is organized as follows: section 2 presents the rationale behind using physical interactions for dynamic scheduling while section 3 focuses on physical background and implementation details. Section 4 presents a set of experiments that compare our approach with graph partitioning tools and dynamic task schedulers. Concluding remarks and future work are discussed in the last section.

2. TASK SCHEDULING WITH POTENTIALS

We represent the set of cells by a multi-dimensional grid (see fig. 2). Each cell has its computing cost of its own, which can evolve over time. We insert a *virtual representation of a computing unit* on the grid, which we call a *vCore*. We then gather tasks around *vCores*, with a Voronoi tessellation [3], by using each *vCore* as a Voronoi site. Voronoi diagrams insure compact sets and provide geometric stability [20]. They are used in many scientific applications (biology, chemistry, computer graphics, ...).

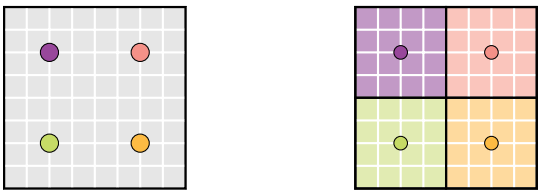

Figure 2: 2D representation of a cells distribution across computing units with a Voronoi tessellation. Every cell represents a task or a graph node and every sphere represents a computing unit. The color of the cell shows the computing unit to which the task is attached to.

These *vCores* are able to *move* on the grid. When moving, *vCores* modify Voronoi domains, which re-equilibrates in turn *vCores* load. Their displacements result of forces computation (with a pair potential): attractive or repulsive forces between *vCores* are balanced by the *vCores* computing load (total cost of Voronoi cell). Overloaded cores are more attractive than underloaded cores, which are then more repulsive.

3. PHYSICAL CONSIDERATIONS

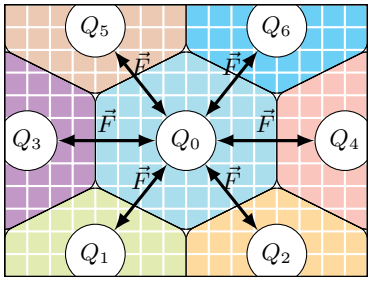

Figure 3: Force interactions between vCores. To compute forces acting on the cell 0, Q charges of neighboring domains are needed.

MD terminology	Analogy	Description
Potential	Potential	Defines forces, and thus system behavior.
Particle	vCore	A computing unit.
Particle charge	vCore load	The load of a computing unit.

Table 1: Analogy summarizes between molecular dynamic simulation and cell scheduling.

Molecular Dynamic (MD) describes the evolution of a set of particles (a system) over time by computing forces between particles, velocities and displacements. The behavior of the whole system is defined by a *potential* that characterizes interactions between particles and, once derived, to forces between particles. In the following, we make an analogy between MD and task scheduling/graph partitioning (a cell can be either a task or a graph node). Table 1 summarizes this analogy.

The computing load Q_i of a *vCore* is the sum of the cost q_k of all the cells in its Voronoi domain (see eq. 1). Hence, a slight cell cost variation has a direct influence on the corresponding *vCore* load.

$$Q_i = \sum_{k \in Voronoi(i)} q_k. \tag{1}$$

Based on this load information, we define forces between *vCores* (see fig. 3 and the next section), and then, we move *vCores*. This point is described in section 3.2, while the next section gives physical background used to compute *vCores* displacements.

3.1 Evolution of a set of charged particles

Let us consider the Coulomb force between two particles i, j separated by a vector $\mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$ with electrical charge C_i, C_j:

$$\mathbf{F}_{ij} = \lambda \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}. \text{ with } \lambda = C_i C_j \tag{2}$$

Different charge signs lead to attractive forces whereas charges with the same sign produce repulsive ones. Pairs interactions of this N-Body system are calculated by exploiting the symmetry of interactions, i.e., $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$. In order to get the relaxed state only, we minimize the potential by using a *steepest descent* algorithm: $\mathrm{d}\mathbf{x}/\mathrm{d}t = -\alpha \mathbf{F}_{ij}$ (with α a positive scalar), and by lumping α and time increment into a

simple scalar k:

$$\mathbf{x_i}(t) = \mathbf{x_i}(t-1) - k \sum_j \mathbf{F}_{ij} \quad (3)$$

We rescale k for every step so that the distance $\mathbf{x}(t) - \mathbf{x}(t-1)$ is a fraction of the cell box dimension, which ensures convergence to stable or metastable states.

3.2 Force

Our potential should minimize the overall computing time. For this purpose, we define the optimal *vCore* load m as the sum of all *vCores* load divided by their number (this choice is discussed in section 5):

$$m = \frac{1}{N} \sum_{i=1,N} Q_i \text{ with } N, \text{ the number of VCores.} \quad (4)$$

The difference $m - Q_i$ thus represents the distance between the load of a particle i and the average (optimal) load m. For two interacting *VCores* i and j, we take a force proportional to the sum of two distances ($m - Q_i$ and $m - Q_j$) instead of the product of charges $C_i C_j$.

$$\mathbf{F}_{ij} = \lambda \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3} + \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^5} \text{ with } \lambda = 1 - \frac{Q_i + Q_j}{2m}. \quad (5)$$

This potential works in this way: an underloaded *vCore* repulses neighboring *vCores* through repulsive forces. Thus, its own Voronoi domain will grow by collecting cells. In the same way, when a *vCore* is overloaded, it has an attractive force, leading other *vCores* to grab cells from it. The last term of equation 5 is used as a short-range repulsive term preventing dipole formation (see details in [17]).

3.3 Iterative model

Since we need to compute particle motions, we face an iterative model. Thus, we have introduced two input parameters, P and I, that respectively characterize the requested precision and the maximum number of iterations used to reach this precision. The precision is a percentage that represents the maximum allowed imbalance. This interface allows a fine-grained usage of the scheduler which permits us to compute a task scheduling only when *VCores* are unbalanced, during the idle time of computing units (see 4.3 for details).

Figure 4 shows that after 100 iterations, the benefit per iteration is close to zero. Therefore, trying to reach a better task distribution (i.e., trying to reduce P) is pointless. Thus a trade-off between P and I must be found. Section 4.3 shows the way we solve this problem.

3.4 Oscillations

In the following, we analyse the relevance of our optimisations and implementation by considering the variance of the *vCore* load, an optimal *vCores* configuration being characterized by a null variance.

Slight oscillations near optimal configuration are visible (see figure 4). These oscillations are due to cells at Voronoi boundaries that can throw out of boundary zone for small variation of *vCore* location. In order to avoid this, we weight the cost of cells by a scalar that decreases with the distance from the boundary of the Voronoi domain (see eq. 6 and fig. 5

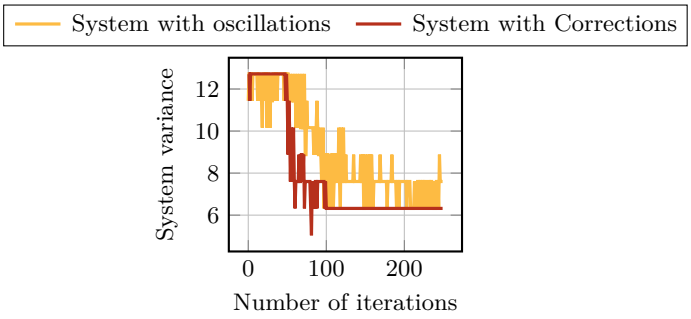

Figure 4: System convergence with oscillations and with oscillation corrections versus the number of iterations for a typical system (50x50 tasks with 64 *vCores*). This plot of the variance of the system shows instabilities when we are close to an optimal configuration (yellow curves). Red curve represents the same situation, but with an oscillation corrections.

for the plotted curve). This implies that cells at boundaries will have a lower weight than cells close to the Voronoi domain center, leading to a more stable Voronoi site. For a given cell i and its two closets Voronoi site A and B, we define the weighted cost, $f(Q)$, defined as:

$$f(Q) = Q \cdot \frac{1}{2}\left[1 - \tanh(k\tilde{\mathbf{AB}} \cdot \tilde{\mathbf{Ai}} - \frac{1}{2}|\tilde{\mathbf{AB}}|)\right] \quad (6)$$

In this equation, the scalar k defines the width of the transition area, i.e, the width of the boundary in which cells are smoothed. For our needs, we simply set $k = 1$, so that only one row of cells is smoothed. Figure 4 shows the affect on oscillations. Therefore, equation 5 becomes:

$$\mathbf{F}_{ij} = \lambda \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3} + \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^5} \text{ with } \lambda = 1 - \frac{\sum_{k=0}^{i} f(Q_i) + \sum_{k=0}^{j} f(Q_j)}{2m}. \quad (7)$$

3.5 Voronoi diagrams

The most time-consuming part of our implementation lies in the Voronoi diagram construction. Voronoi diagrams are computed frequently, since every partitioning triggers a new diagram construction. Since our algorithm actually computes Voronoi sites which compose the whole diagram, we do not need to rely on an external voronoi diagram construction library [5, 7, 23]. We only need to attach each cell to a site in order to create Voronoi domains. Hence, we need to determine, for each cell, the closest *vCore*. A *naive* implementation is in $\mathcal{O}(n \cdot m)$ (n, the number of cells, m, the number of *vCores*), but several more efficient nearest-neighbor-search algorithms obviously exist [4, 14, 6]. The Vantage-Point tree [24], which has good construction ($\mathcal{O}(n \cdot log(n))$ and search performance ($\mathcal{O}(log(n))$ is considered in the following.

4. RESULTS

To evaluate our scheduler, we have developed two representative stencil-like model applications. The first one, uses

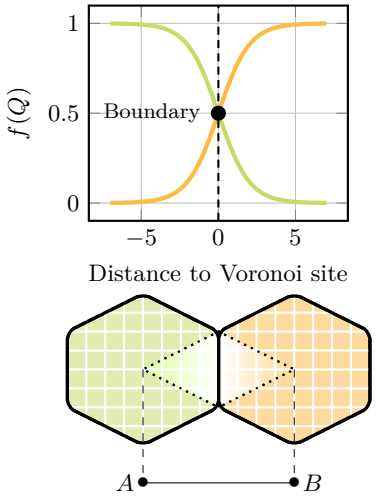

Figure 5: Voronoi boundary cell cost smoothing as a function of distance.

a virtual cells domain, where every cell has a given cost generated by a random Perlin noise [19]. We don't do any computation with cells. This allows us to compare the quality and the time needed for the partition of the domain (see 4.1.1). The second one uses a real cells domain and provides an intensive memory usage with data exchanges between cells. Every cell has 4 direct neighbors (four cardinal directions), and needs data (a matrix of `double`) from its neighbors to compute its own matrix. These exchanged matrices are the result of the previous iteration. In a distributed memory (see section4.1.2), this allows us to evaluate the number of `MPI` communications, induced after the new domain decomposition and, during computation (data exchange between meshes). We evaluate our scheduler with sequential partitioning tools and parallel partitioning tools (with partitioning refinement enabled). Section 4.2 evaluate the same application, but in a shared memory, by comparing our scheduler against common task scheduling strategies. Thus, our analysis focuses on cache miss rates.

4.1 Comparison with Graph Partitioners

4.1.1 Scheduling quality & efficiency

This section focuses on the time needed to compute one partition, and the quality of the resulting partition. For this specific purpose, we compare our scheduler against `Scotch` [18] (`LaBRI/INRIA` Bordeaux Sud-Ouest) and `Metis` [13] (University of Minnesota). With `Spawn`, we compute the domain of each *vCores*, and with `Scotch` or `Metis`, we compute a partition with [*number of vCores*] parts. For every single configuration (dimension of the grid, number of *vCores*), we run thousands of sequential tests on a E5-2650 Intel® Xeon (2.0 GHz) processor. Figure 6 shows an example of generated Perlin noise along with different generated partitions, in terms of shape, after a call to each of these tools. Figure 7 shows results for a grid of 256x256 cells, with a variable number of *vCores*. For our algorithm, we use $P = 20$ and $I = 20$. These two charts represent the time (x-axis) needed to compute a partition of a given quality (y-axis). The quality of the partition is the distance (in percentage) to the optimal configuration in the interval $[optimal, 2 \times optimal]$.

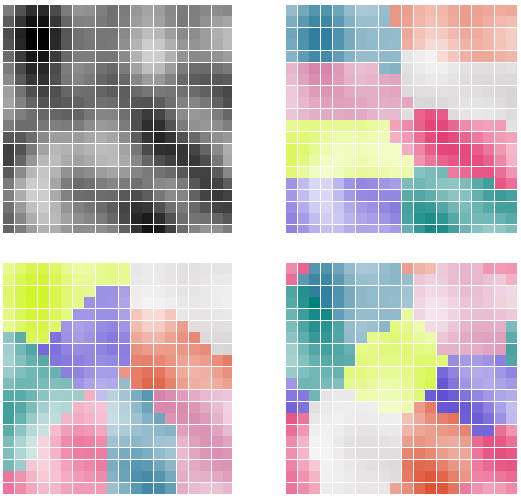

Figure 6: Inital random Perlin noise (top-left), cells distribution with `Metis` (top-right), cells distribution with `Scotch` (bottom-left) and cells distribution with `Spawn` (bottom-right).

The left chart shows the time needed to compute a partition for a given initial configuration, while the right chart represents the time needed to compute a partition after a load variation on the task domain. `Scotch` and `Metis` perform at least twice faster than `Spawn` when computing the initial partition. This is related to our iterative model and to the initial *vCores* position that can be far from the optimal position. On the other hand (right chart), `Spawn` shows better results after a load variation. This is a consequence of the model, we use the previous decomposition to compute the new one. Thus, if the previous decomposition is relatively close to the new optimal, after a load variation, we need a few iterations to reach the new optimal task distribution. If we decrease the P parameter (in order to reach a better task distribution), the benefit vanishes and requires a higher computing time (see bottom chart of figure 7).

We can notice an important computing time and a far-from-optimal partitioning when using 512 *vCore*. We explain this behavior by a over demanding P parameter, 20% in this case, with respect to the configuration. Since we use a Voronoi tessellation, when *vCores* move, they exchange more than one task, and possibly a full boundary. Consequently, the 20% imbalance parameter represents a cost that is weaker than the cost *vCores* may exchange. Figure 8 confirms this observation: we compare the needed time (y-axis) to reach a given requested imbalance degree (x-axis), within at most 1,000 iterations (I parameter). We can see that for the same number of *vCores*, but for a variable domain size, we are unable to go further a given imbalance. In conclusion, this approach produces a task distribution with a quality comparable to the one produced by `Scotch` or `Metis`, with a slight overhead. However, due to the geometric shape of cells, leading the optimal task distribution might be difficult in regard to the system configuration (size and number of *vCores*) and to input parameters P and I. Nevertheless, our way of using our scheduler (see sec. 4.2) tackles this limitation since we are able to compute task distribution in

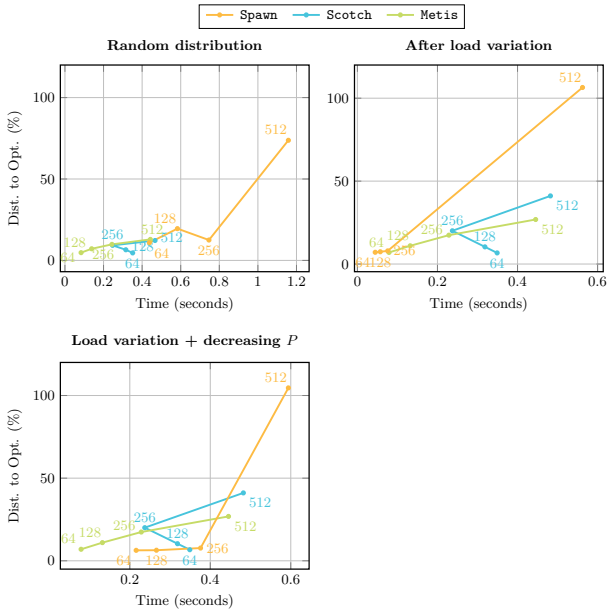

Figure 7: Efficiency & quality domain partitioning comparison with a random load distribution (top-left), after a load variation (top-right) and after a load variation plus a modification of the P parameter (bottom-left). Two graph partitioning tools are used (`Metis` and `Scotch`) and compared with `Spawn`. These graphics show the time needed to compute the partition of a 256x256 domain of tasks, with a variable number of *vCores* (or parts).

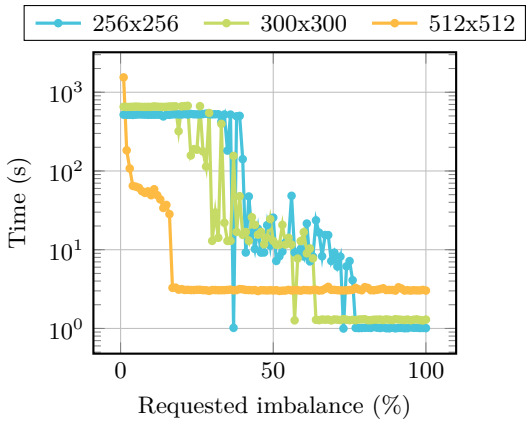

Figure 8: Time needed to reach a given *vCores* imbalance, within at most 1,000 iterations, for different domain sizes. 0% of imbalance means perfect task distribution while a 100% of imbalance means that we can have one cell that has at most twice the optimal load.

background and as long as an imbalance exists.

4.1.2 *Performance*

This section focuses on performance of `Spawn`, in a dynamic task scheduling context, by using an iterative model application (as described in the beginning of section 4). For both partitioning tools and `Spawn` usage, a new partition is computed between each new iteration, with newest task costs information. Since such tools use graphs as input, we use tasks as graph vertices, and communication pattern between tasks as edges. For both tools, we use a criterion that produces compact sets of vertices, with an equilibrium between the load of sets.

Shared Memory. Figure 9 compares performance of `Spawn` in relation to `Scotch` [18] and `Metis` [13] libraries. Left figures refer to the static load case while right charts refer to the dynamic load case, and, upper charts refer to the speedup while the bottom ones refer to the average number of L1 data cache misses per threads for one iteration. Here, all partitioning strategies produce compact sets, minimizing data transfers between tasks, but `Spawn` has better performance in both static and dynamic load variations. As explained in section 1, `Scotch` and `Metis` produce a new and totally different task distribution after a task costs variation. Due to hardware and Operating System behavior, a task cost may vary over time, even if it's the same task and it does the same things. Hence, such partitioning tools produce different task distributions after each iteration. This involves data displacements between sockets and higher cache miss rates (see bottom charts in fig: 9).

Furthermore, in section 4.1.1, we saw that `Spawn` has a higher computing cost. Figure 9 puts in evidence that this cost is fully hidden by the quality and benefits of the resulting scheduling.

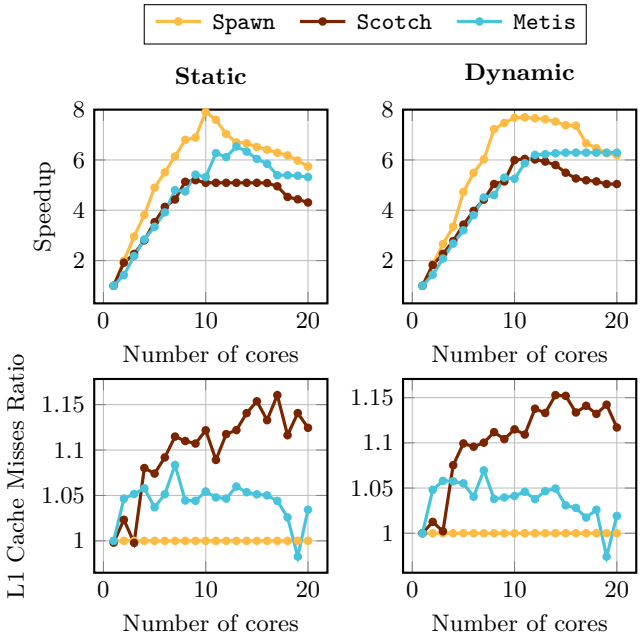

Figure 9: Resulting speedups and L1 data cache misses with partitioning tools (`Scotch`, `Metis`) and with `Spawn` on an IvyBridge Xeon E5-2680 computing node.

Distributed Memory. Figures 10 and 11 present the same evaluation but in a distributed memory environment (using `MPI` libraries). Measurements focus on speedup and data transfer rate between each iteration. We use matrices of 128x128 `doubles` in a domain of 512x512 tasks (262,144 tasks). Tests run on a cluster of Intel® Xeon X7650 nodes, interconnected with an Infiniband QDR network. Figure 10 shows the average number of exchanged tasks on the `MPI` network (in percentage of the entire domain) and the resulting application speedup. `Scotch` and `Metis` involve an important data exchange since they aren't designed for dynamic partitioning: nearly the entire task domain is exchanged for each iteration while less than 5% of the domain is exchanged with `Spawn`.

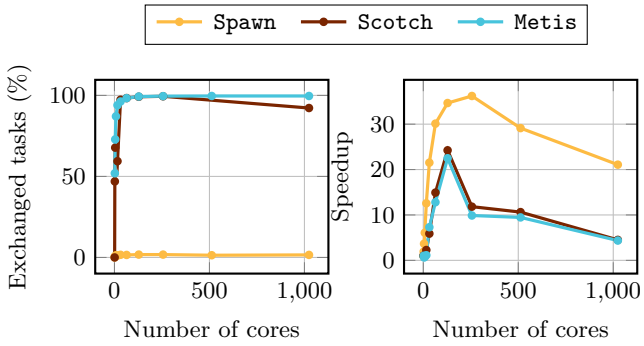

Figure 10: Resulting speedups and data transfer rates with partitioning tools (`Scotch`, `Metis`) and with `Spawn`.

Parallel and Refinement-Capable Graph Partitioners. Tools like `PTScotch` [8] and `ParMETIS` [12] can perform parallel graph partitioning. `ParMETIS` supports refinement graph partitioning, i.e., it can take into account the actual partition to produce a new partition with the newest task costs. `Zoltan` [9] is another partitioning tool that can use not only its internal algorithms but also `Scotch`/`PTScotch` or `ParMETIS` to compute a graph partitioning.

One limitation of these tools is that they need to know, for every local tasks and for every neighboring local or remote task, the identifier of the node that owns it. This implies to maintain on each `MPI` node, and with every call to a partitioning routine, the list of neighboring nodes, and the list of tasks they share between them.

Next evaluation focuses on `Zoltan` and `ParMETIS` (with `Zoltan`) tools. We evaluate `Zoltan` with `PHG` and `RCB` partitioning algorithms. `PHG` is the internal Parallel Hypegraph and Graph partitioning method. It supports initial partitioning, refinement and re-partitioning operations. Both repartitioning and refinement operations reuse the current partition, but refinement operation is stricter. The `RCB` method is a geometric cutting algorithm that recursively divide the computational domain in two sub-domains, of equal work load, until the number of sub-domains is equal to the number of requested parts. We use one internal algorithm option, `RCB_REUSE`, that indicates whether previous cuts should be used as initial guesses for the current `RCB` partition.

Figure 11 shows data transfer rates and speedups of `Zoltan` (with `PHG`, `RCB` and `ParMETIS`) in comparison with `Spawn`. We can see that `Spawn` performance are similar to graph partitioning and geometric algorithms when the network isn't used (i.e. one computing node used). For a few number of `MPI` used nodes, `Spawn` is still better than `Zoltan` or `ParMETIS`, while, for a higher number of `MPI` nodes, `RCB` algorithm outperforms others algorithms. In a more detailed point of view, if we compare the `RCB` algorithm to `Spawn`, we can see (left chart), that `Spawn` produces less data transfers (1-3% of the entire domain versus 5-35% for `RCB`). This explains that our algorithm involves a better speedup when the network isn't used (time spent in internal communication is regained by the time *not* spent in exchanging data). On the other hand, with a higher number of nodes, central behavior of `Spawn` is penalizing and collective `MPI` operations considerably slow down the global computation.

This observation is, however, incorrect if we compare `Spawn` to the `PHG` method or to `ParMETIS`. In comparison with `Spawn`, data transfer rates are more important for `PHG`, and really similar for `ParMETIS`. Thus, we could expect better results for, at least, `ParMETIS`. By considering that partitioning tools have scalability limitations (because of internal synchronizations, exacerbate by the network), we can consider that data transfers are less penalizing than scalability limitations, which explain `PHG` and `ParMETIS` results shown in figure 11.

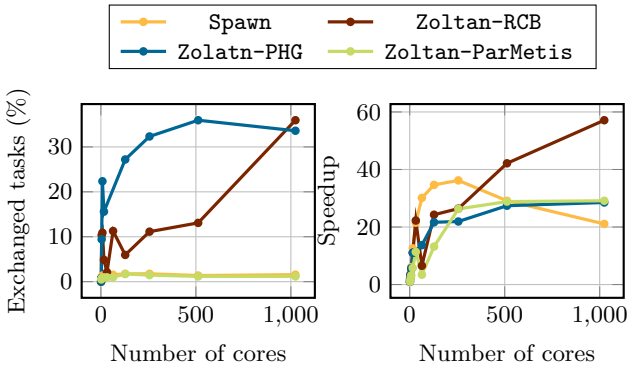

Figure 11: Resulting speedups and data transfer rates with `Zoltan` (`RCB`, `PHG`), `ParMetis` (through `Zoltan`) and with `Spawn`.

4.2 Comparison with Common Scheduling Strategies

We have extended the StarPU [2] runtime (the development version, see the next section for details) with our scheduler to evaluate `Spawn` performance against common task scheduler. Data block of tasks are managed by StarPU through the `starpu_block_data_register` routine, which enables automatic and backgrounded data transfers. We evaluate our scheduler on a two-sockets Intel® Xeon E5-2680 (Ivy-Bridge, 2x10 cores, 2.8GHz) machine. We did a hundred of runs with a domain of 100x100 tasks, where each task contains a matrix of 8x8 `doubles`. We evaluate performance in both static and dynamic load variations over time.

4.3 Integration

`StarPU` comes with a variety of scheduling policies, but also offers the capability to define new ones. This allows us to manage `StarPU` workers (abstract representation of computing units) ourselves. `Spawn` is used by `StarPU` through an intermediate *meta-scheduler* that implements interfaces required by `StarPU` (the `starpu_sched_policy` C-structure). When tasks are submitted to `StarPU`, they are forwarded to the scheduling layer, and so, to our meta-scheduler. This meta-scheduler `asks` our library (`Spawn`), to which worker this task should be assigned to (more precisely, to which Voronoi domain this task is attached to).

If it is the first time that our library has been called, then it hasn't any information on tasks, therefore, one initial distribution is used. This initial distribution depends on the initial position of *vCores*, which can be, for example, a regular grid or a random position. Once tasks are distributed among `StarPU` workers, their execution (i.e. the computing time) is monitored and then sent to our library.

The scheduling can then be achieved in an imperceptible way for the final user (see Fig. 12): it is computed by the first StarPU worker that has finished its own set of tasks, and does it as long as another worker is still working. Thus, we only compute a task distribution when an imbalance appears. This has also the advantage to allow us to set $P = 1$ and $I = 1$ as parameters of our algorithm (which can be interpreted as *"as long as an imbalance exists, do one iteration to reach the most optimal distribution"*). As explained, a new task distribution is based on the task information (i.e. computing cost) of the previous tasks execution. Therefore, the first two load balancing steps might not be optimal.

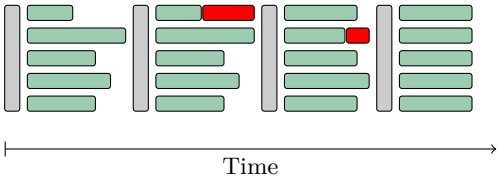

Figure 12: Timeline representing StarPU calls (gray), task computation (green) and partitioning computation with `Spawn` (red). Partitioning computation is done only when an imbalance is present and continues as long as a StarPU worker has ongoing tasks.

4.4 Results

Figure 13 compares performance of `Spawn` in relation to internal StarPU schedulers. Left figures refer to the static load case while right figures refer to the dynamic load case. In the same way, upper charts refer to the speedup while the bottom ones refer to the ratio of L1 data cache misses per thread for one iteration in comparison with `Spawn`. We use three internal StarPU schedulers: `Eager`, `DM` and `DMDA`. `Eager` uses a simple FIFO-based greedy policy to schedule tasks over computing units. `DM` and `DMDA` use performance models allowing them to perform smart task placement over StarPU workers, with the objective to minimize the overall execution time. `DMDA` works like `DM`, but takes into account

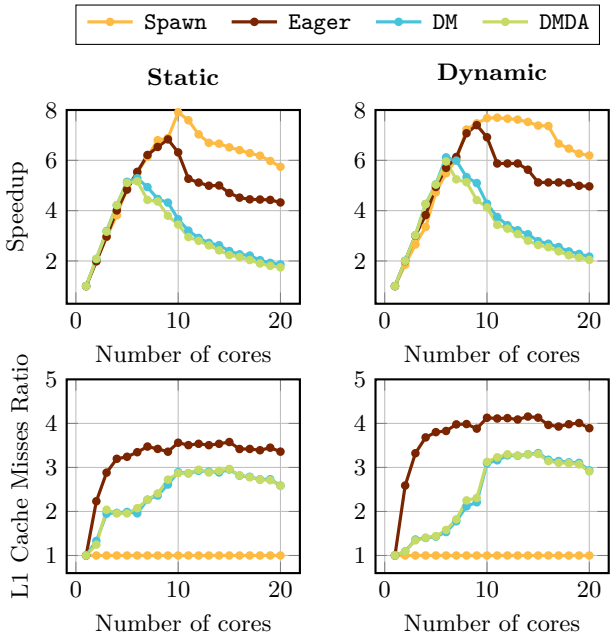

Figure 13: Resulting speedups and L1 data cache misses with internal StarPU schedulers (`Eager`, `DM`, `DMDA`) and with `Spawn` on an IvyBridge Xeon E5-2680 computing node.

data transfer time during task placement. More information on this can be found in the StarPU HandBook [22].

In both static and dynamic load variation cases, `Spawn` is close to StarPU schedulers performance and outperforms it with the growth of the number of threads. Bottom parts of figures show that `Spawn` dramatically reduces the number of `L1` cache misses with a factor between two and four, thus explaining the gain. This is due to the `Spawn` ability to provide compact sets of tasks, reducing data transfers between threads.

On the other side, if we compare task cache misses with `Eager`, `DM` and `DMDA`, we note that `Eager` doubles the number of cache misses that `DM` and `DMDA` induce, but has better performance (see the speedup in fig. 13). This is related to the behavior of these schedulers: by using a task queue, `Eager` naturally maximizes StarPU workers activity, in opposition to `DM` and `DMDA` that distribute tasks *in advance* as soon as they are ready in order to minimize overall execution time. Since our application uses intensive memory accesses, the execution time is strongly related to memory accesses, and task execution time depends on the *place* where tasks are executed. Therefore, assigned tasks by `DM` or `DMDA` constitute a set of non-related tasks, like they would have been assigned with `Eager`. Hence, with `DM` or `DMDA`, StarPU workers are imbalanced, since the execution time of assigned tasks doesn't reflect their real execution time. A deeper analysis of StarPu execution traces (thanks to the `StarPU` support of FxT[1] traces generation) (see fig 14) confirms this point. With this information, it appears that common tasks schedulers seem unable to perform well with such task configurations (neighboring and data dependencies).

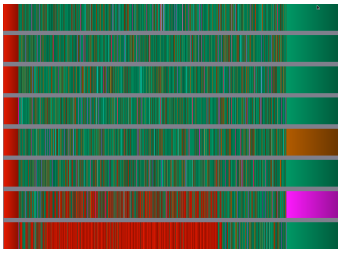

Figure 14: FxT traces with the `DM` scheduler. Green elements refer to tasks execution while red elements refer to inactivity of StarPU workers. One row symbolise the activity of one StarPU worker.

5. DISCUSSION

Our force equation (see eq. 5) defines the optimal *vCore* load m as the sum of all *vCores* load divided by their number. This centralized characteristic implies distinct considerations depending on whether we are in a shared or in a distributed memory environment. In a shared memory environment, we need to store task costs. Depending on the size of the domain and the number of used `vCores`, it can imply an important memory cost. In a distributed memory environment, this implies to retrieve all the task costs information on one node. Beside the fact that this requires one ALL-to-ONE communication each time we do a partitioning, this also necessitates a lot of memory into one node to store all the information.

Despite these considerations, results presented in sections 4.1.2 and 4.2 show us interesting performance. An improved version of our algorithms, with less centralized properties, should scale well on multiple nodes.

6. CONCLUSION & FUTURE WORK

This Pair-Potential approach for task scheduling shows good scheduling properties. Data locality and dynamic load variations are well handled, for a small computing cost. In a shared memory, or with a few number of MPI nodes, experiments show that this method is really comparable and outperforms well known graph partitioners and common task schedulers. Nevertheless, the current implementation has some limitations. Central characteristics of the current algorithm involves a memory consumption and communications that increase with the number of computing units (cores, `MPI` nodes). Therefore, the current implementation isn't well scalable on a large number of MPI nodes. Our next work will thus focus on a parallel version of the algorithm, for both shared and distributed memory, with a distributed information, and an inter-nodes force computation.

7. REFERENCES

[1] Fxt library for execution traces generation. '`https://savannah.nongnu.org/projects/fkt/`'.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] Franz Aurenhammer and Rolf Klein. Voronoi diagrams. *Handbook of Computational Geometry*, pages 201–290, 2000.

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[5] Boost. Boost c++ libraries. "`http://www.boost.org/doc/libs/1_57_0/libs/polygon/doc/voronoi_main.htm`".

[6] Boost. Boost c++ libraries. "`http://www.boost.org/doc/libs/1_57_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html`".

[7] CGal. Cgal - voronoi diagrams. "`http://doc.cgal.org/latest/Manual/packages.html#PartVoronoiDiagrams`".

[8] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. In *4th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'06)*, Rennes, France, September 2006. Extended abstract, 2 pages.

[9] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John, and Courtenay Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proc. Intl. Conf. on Supercomputing*, pages 110–118, Santa Fe, New Mexico, 2000.

[10] Mozzila foundation. The rust programming language. '`http://www.rust-lang.org/`'.

[11] Scott Hemmert. Green hpc: From nice to necessity. *Computing in Science & Engineering*, 12(6):0008–10, 2010.

[12] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[13] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[14] Joseph Kuan and Paul Lewis. Fast k nearest neighbour search for r-tree family. In *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, pages 924–928. IEEE, 1997.

[15] Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[16] OpenMP-Committee. Openmp application program interface 3.0. Technical report.

[17] Jean-Charles Papin, Christophe Denoual, Laurent Colombet, and Raymond Namyst. Dynamic load balancing with pair potentials. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, pages 462–473, 2014.

[18] François Pellegrini. Scotch and libScotch 5.1 User's Guide, August 2008. 127 pages.

[19] Ken Perlin. Perlin noise is a computer-generated visual effect developed by ken perlin, who won an academy award for technical achievement for inventing it.

[20] Daniel Reem. The geometric stability of voronoi

diagrams with respect to small changes of the sites. In *Proceedings of the Twenty-seventh Annual Symposium on Computational Geometry*, SoCG '11, pages 254–263, New York, NY, USA, 2011. ACM.

[21] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, November 2001.

[22] INRIA Runtime team. Starpu handbook. "`http://starpu.gforge.inria.fr/doc/starpu.pdf`".

[23] Voro++. Voro++ - a 3d voronoi cell software library. "`http://math.lbl.gov/voro++/`".

[24] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.