Partitioning Spatially Located Computations using Rectangles

Erik Saule*, Erdeniz Ö. Baş*† and Ümit V. Çatalyürek*‡
* *Department of Biomedical Informatics*
† *Department of Computer Science and Engineering*
‡ *Department of Electrical and Computer Engineering*
The Ohio State University
Email: {esaule,erdeniz,umit}@bmi.osu.edu

Abstract

The ideal distribution of spatially located heterogeneous workloads is an important problem to address in parallel scientific computing. We investigate the problem of partitioning such workloads (represented as a matrix of positive integers) into rectangles, such that the load of the most loaded rectangle (processor) is minimized. Since finding the optimal arbitrary rectangle-based partition is an NP-hard problem, we investigate particular classes of solutions, namely, rectilinear partitions, jagged partitions and hierarchical partitions. We present a new class of solutions called m-way jagged partitions, propose new optimal algorithms for m-way jagged partitions and hierarchical partitions, propose new heuristic algorithms, and provide worst case performance analyses for some existing and new heuristics. Moreover, the algorithms are tested in simulation on a wide set of instances. Results show that two of the algorithms we introduce lead to a much better load balance than the state-of-the-art algorithms.

1. Introduction

The key to obtaining good efficiency in parallel and distributed computing is to ensure that the data and hence relevant computations of a parallel application are distributed in a load balanced fashion while keeping the communication volume low. In a large class of applications, the computational tasks are located in a discrete, two or three-dimensional space and each task only communicates with its neighboring tasks. That class of applications includes linear algebra kernels [1], [2], [3], image rendering algorithms [4] and particle-in-cell simulation [5], [6] (used in fluid dynamics, weather forecast, magnetic field simulation, and so on).

To distribute such applications on a parallel computer, one must decide on which processor each cell (voxel) of the two (three) dimensional space will be processed, provided the computational requirement of each cell. Since many scientific applications exhibit localized computations, preferably, each processor should be allocated a connected and compact region of the computational work space, to reduce the communication volume. Previous literature has investigated different shapes such as triangles or ovoids. However, rectangles (and rectangular volumes) are the most preferred shape since they implicitly minimize the communication while not restricting the set of possible allocations drastically. Moreover, their compact representation also allows to easily find which processor a given cell is allocated to.

This paper addresses the problem of partitioning a load matrix into a given number of rectangles to minimize the load of the most loaded rectangle. Computing the optimal solution of this problem is NP-Hard. Therefore, like many of the previous research, we focus on designing faster algorithms by restricting our search to specific classes of solutions. Most of the previous studies only consider a limited set of classes, a reduced set of algorithms, and either only theoretical validation or practical validation on limited test cases. In this paper, we attempt to re-classify rectangle partitions, present optimum algorithm and theoretically-sound heuristics for different classes of partitions, and experimentally evaluate those algorithms across various kinds of applications. We would like to emphasize that, in some classes despite the fact that an optimal partition can be found in polynomial time, the practical runtime might be too high (because of the high-order polynomial complexity) hence making those solutions impractical. Therefore, we also investigate heuristic algorithms in such cases.

The contributions of this work are as follows. A classical $P \times Q$-way jagged heuristic is theoretically analyzed by bounding the load imbalance it generates in the worst case. We propose a new class of solutions, namely, m-way jagged partitions, for which we propose a fast heuristic as well as an exact polynomial dynamic programming formulation. This heuristic is also theoretically analyzed and shown to perform better than the $P \times Q$-way jagged heuristic. For an existing class of solutions, namely, hierarchical bipartitions, we propose both an optimal polynomial dynamic programming algorithm as well as a new heuristic. The presented and proposed algorithms are practically assessed in simulations performed on synthetic load matrices and on real load matrices extracted from both a particle-in-cell

This work was supported in parts by the U.S. DOE SciDAC Institute Grant DE-FC02-06ER2775; by the U.S. National Science Foundation under Grants CNS-0643969, OCI-0904809 and OCI-0904802.

simulator and a geometric mesh. Simulations show that both the proposed heuristics outperform all the tested existing algorithms.

Similar classes of solutions are used in the problem of partitioning a equally loaded tasks onto heterogeneous processors (see [7] for a survey). This problem often assumes the task space is continuous (therefore infinitely divisible). Since the load balance is trivial to optimize in such a context, most work in this area focus on optimizing communication patterns.

The rest of the paper is organized as follows. Section 2 presents the model and notations used. The different classes of partitions and known algorithms to generate them are described in Section 3. This section also presents new optimal and heuristic polynomial time algorithms for a large class of rectangle partitions. The algorithms are evaluated in Section 4 on synthetic dataset as well as on dataset extracted from two real simulation codes. Conclusive remarks are presented in Section 5.

2. Model and Preliminaries

2.1. Problem Definition

Let A be a two dimensional array of $n_1 \times n_2$ positive integers representing the spatially located load. This load matrix needs to be distributed on m processors. Each element of the array must be allocated to exactly one processor. The load of a processor is the sum of the elements of the array it has been allocated. The cost of a solution is the load of the most loaded processor. The problem is to find a solution that minimizes the cost.

In this paper we are only interested in rectangular allocations, and we will use 'rectangle' and 'processor' interchangeably. That is to say, a solution is a set S of m rectangles $r_i = (x_1, x_2, y_1, y_2)$ which form a partition of the elements of the array. Two properties have to be ensured for a solution to be valid: $\bigcap_{r \in R} = \emptyset$ and $\bigcup_{r \in R} = A$. The first one can be checked by verifying that no rectangle collides with another one, it can be done using line to line tests and inclusion test. The second one can be checked by verifying that all the rectangles are inside A and that the sum of their area is equal to the area of A. This testing method runs in $O(m^2)$. The load of a processor is $L(r_i) = \sum_{x_1 \leq x \leq x_2} \sum_{y_1 \leq y \leq y_2} A[x][y]$. The load of the most loaded processor in solution S is $L_{max} = \max_{r_i} L(r_i)$. We will denote by L^*_{max} the minimal maximum load achievable. Notice that $L^*_{max} \geq \frac{\sum_{x,y} A[x][y]}{m}$ and $L^*_{max} \geq \max_{x,y} A[x][y]$ are lower bounds of the optimal maximum load. In term of distributed computing, it is important to remark that this model is only concerned by computation times and not by communication times.

Algorithms that tackle this problem rarely consider the load of a single element of the matrix. Instead, they usually consider the load of a rectangle. Therefore, we assume that matrix A is given as a 2D prefix sum array Γ so that $\Gamma[x][y] = \sum_{x' \leq x, y' \leq y} A[x'][y']$. That way, the load of a rectangle $r = (x_1, x_2, y_1, y_2)$ can be computed in $O(1)$ (instead of $O((x_2 - x_1)(y_2 - y_1))$), as $L(r) = \Gamma[x_2][y_2] - \Gamma[x_1 - 1][y_2] - \Gamma[x_2][y_1 - 1] + \Gamma[x_1 - 1][y_1 - 1]$.

An algorithm H is said to be a ρ-approximation algorithm, if for all instances of the problem, it returns a solution which whose maximum load is no more than ρ times the optimal maximum load, i.e., $L_{max}(H) \leq \rho L^*_{max}$. In simulations, the metric used for qualifying the solution is the load imbalance which is computed as $\frac{L_{max}}{L_{avg}} - 1$ where $L_{avg} = \frac{\sum_{x,y} A[x][y]}{m}$. A solution which is perfectly balanced achieves a load imbalance of 0. Notice that the optimal solution for the maximum load might not be perfectly balanced and usually has a strictly positive load imbalance. The ratio of most approximation algorithm are proved using L_{avg} as the only lower bound on the optimal maximum load. Therefore, it usually means that a ρ-approximation algorithm leads to a solution whose load imbalance is less than $\rho - 1$.

2.2. The One Dimensional Variant

Solving the 2D partitioning problem is obviously harder than solving the 1D partitioning problem. Most of the algorithms for the 2D partitioning problems are inspired by 1D partitioning algorithms. A theoretical and experimental comparison of those algorithms has been given in [8]. In [8], the fastest optimal 1D partitioning algorithm is `NicolPlus`; it is an algorithmically engineered modification of [9], which uses a subroutine proposed in [10]. A slower optimal algorithm using dynamic programming was proposed in [11]. Different heuristics have also been developed [12], [8]. Frederickson [13] proposed an $O(n)$ optimal algorithm which is only arguably better than $O((m \log \frac{n}{m})^2)$ obtained by `NicolPlus`. Moreover, Frederickson's algorithm requires complicated data structures which are difficult to implement and are likely to run slowly in practice. Therefore, in the remainder of the paper `NicolPlus` is the algorithm used for solving one dimensional partitioning problems.

In the one dimensional case, the problem is to partition the array A composed of n positive integers into m intervals.

`DirectCut (DC)` (called "Heuristic 1" in [12]) is the fastest reasonable heuristic. It greedily allocates to each processor the smallest interval $I = \{0, \ldots, i\}$ which load is more than $\frac{\sum_i A[i]}{m}$. This can be done in $O(m \log \frac{n}{m})$ using binary search on the prefix sum array and the slicing technique of [10]. By construction, `DC` is a 2-approximation algorithm but more precisely, $L_{max}(DC) \leq \frac{\sum_i A[i]}{m} + \max_i A[i]$. This result is particularly important since it provides an upper bound on the optimal maximum load: $L^*_{max} \leq \frac{\sum_i A[i]}{m} + \max_i A[i]$.

A widely known heuristic is `Recursive Bisection (RB)` which recursively splits the array into two parts of

similar load and allocates half the processors to each part. This algorithm leads to a solution such that $L_{max}(RB) \leq \frac{\sum_i A[i]}{m} + \max_i A[i]$ and therefore is a 2-approximation algorithm [8]. It has a runtime complexity of $O(m \log n)$.

The optimal solution can be computed using dynamic programming [11]. The formulation comes from the property of the problem that one interval must finish at index n. Then, the maximum load is either given by this interval or by the maximum load of the previous intervals. In other words, $L^*_{max}(n,m) = \min_{0 \leq k < n} \max L^*_{max}(k, m-1), L(\{k+1, \ldots, n\})$. A detailed analysis shows that this formulation leads to an algorithm of complexity $O(m(n-m))$.

The optimal algorithm in [9] relies on the parametric search algorithm proposed in [10]. A function called *Probe* is given a targeted maximum load and either returns a partition that reaches this maximum load or declares it unreachable. The algorithm greedily allocates to each processor the tasks and stops when the load of the processor will exceed the targeted value. The last task allocated to a processor can be found in $O(\log n)$ using a binary search on the prefix sum array, leading to an algorithm of complexity $O(m \log n)$. [10] remarked that there are m binary searches which look for increasing values in the array. Therefore, by slicing the array in m parts, one binary search can be performed in $O(\log \frac{n}{m})$. It remains to decide in which part to search for. Since there are m parts and the searched values are increasing, it can be done in an amortized $O(1)$. This leads to a *Probe* function of complexity $O(m \log \frac{n}{m})$.

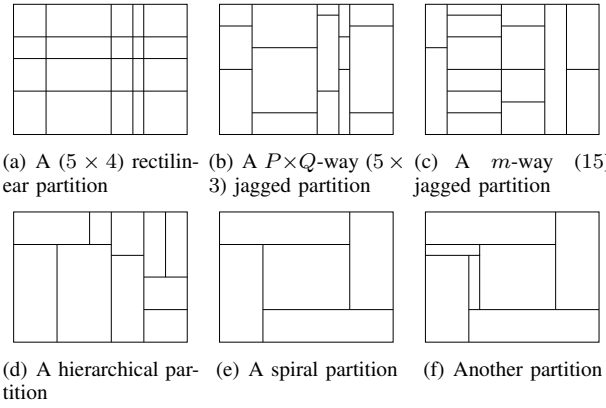
The algorithm proposed by [9] exploits the property that if the maximum load is given by the first interval then its load is given by the smallest interval so that $Probe(L(\{0, \ldots, i\}))$ is true. Otherwise, the largest interval so that $Probe(L(\{0, \ldots, i\}))$ is false can safely be allocated to the first interval. Such an interval can be efficiently found using binary search, and the array slicing technique of [10] can be used to reach a complexity of $O((m \log \frac{n}{m})^2)$. Recent work [8] showed that clever bounding techniques can be applied to reduce the range of the various binary searches inside *Probe* and inside the main function leading to a runtime improvement of several orders of magnitude.

3. Algorithms

This section describes algorithms that can be used to solve the 2D partitioning problem. These algorithms focus on generating a partition with a given structure. The considered structures are presented in Figure 1. Notice that each structure is a generalization of the previous one.

3.1. Rectilinear Partitions

Rectilinear partitions (also called General Block Distribution in [14], [15]) organize the space according to a $P \times Q$ grid as shown in Figure 1(a). This type of partitions is often



(a) A (5×4) rectilinear partition (b) A $P \times Q$-way (5×3) jagged partition (c) A m-way (15) jagged partition

(d) A hierarchical partition (e) A spiral partition (f) Another partition

Figure 1. Different structures of partitions.

used to optimize communication and indexing and has been integrated in the High Performance Fortran standard [16]. It is the kind of partition constructed by the MPI function `MPI_Cart`. This function is often implemented using the `RECT-UNIFORM` algorithm which divides the first dimension and the second dimension into P and Q intervals with size $\frac{n_1}{P}$ and $\frac{n_2}{Q}$ respectively. Notice that `RECT-UNIFORM` returns a naive partition that balances the area and not the load.

Computing the optimal rectilinear partition is shown to be an NP-Hard problem [17]. [14] points out that the NP-completeness proof in [17] implies that there is no $(2-\epsilon)$-approximation algorithm unless P=NP. We can also remark that the proof is valid for a given $P \times Q$ grid, but the complexity of the problem is unclear if the only constraint is that $PQ \leq m$. Notice that, the load matrix is often assumed to be a square.

[9] (and [15] independently) proposed an iterative refinement heuristic algorithm that we call `RECT-NICOL` in the remaining of this paper. Provided the partition in one dimension, called the fixed dimension, `RECT-NICOL` computes the optimal partition in the other dimension using an optimal one dimension partitioning algorithm. The one dimension partitioning problem is built by setting the load of an interval of the problem as the maximum of the load of the interval inside each stripe of the fixed dimension. At each iteration, the partition of one dimension is refined. Each iteration runs in $O(Q(P \log \frac{n_1}{P})^2)$ or $O(P(Q \log \frac{n_2}{Q})^2)$ depending on the refined dimension. According to the analysis in [9] the number of iterations is $O(n_1 n_2)$ in the worst case; however, in practice the convergence is faster (about 3-10 iterations for a 514*514 matrix up to 10,000 processors). [14] shows it is a $\theta(\sqrt{m})$-approximation when $P = Q = \sqrt{m}$.

The first constant approximation algorithm for rectilinear partitions have been proposed by [18] but neither the constant nor the actual complexity is given. [14] claims it is a 120-approximation that runs in $O(n_1 n_2)$.

[14] presents two different modifications of

RECT-NICOL which are both a $\theta(\sqrt{p})$-approximation algorithm for the rectilinear partitioning problem of a $n_1 \times n_1$ matrix in $p \times p$ blocks which therefore is a $\theta(m^{1/4})$-approximation algorithm. They run in a constant number of iterations (2 and 3) and have a complexity of $O(m^{1.5}(\log n)^2)$ and $O(n(\sqrt{m}\log n)^2)$. [14] claims that despite the approximation ratio is not constant, it is better in practice than the algorithm proposed in [18].

[19] provides a 2-approximation algorithm for the rectangle stabbing problems which translates into a 4-approximation algorithm for the rectilinear partitioning problem. This method is of high complexity $O(\log(\sum_{i,j} A[i][j])n_1^{10}n_2^{10})$ and heavily relies on linear programming to derive the result.

[20] considers resource augmentation and proposes a 2-approximation algorithm with slightly more processors than allowed. It can be tuned to obtain a $(4+\epsilon)$-approximation algorithm which runs in $O((n_1+n_2+PQ)P\log(n_1n_2))$.

3.2. Jagged Partitions

Jagged partitions (also called Semi Generalized Block Distribution in [15]) distinguish between the main dimension and the auxiliary dimension. The main dimension will be split in P intervals. Each rectangle of the solution must have its main dimension matching one of these intervals. The auxiliary dimension of each rectangle is arbitrary. Examples of jagged partitions are depicted in Figures 1(b) and 1(c). The layout of jagged partitions also allows to easily know which rectangle contains a given element [3].

Without loss of generality, all the formulas in this section assume that the main dimension is the first dimension.

3.2.1. $P \times Q$-way Jagged Partitions. Traditionally, jagged partition algorithms are used to generate what we call $P \times Q$-way jagged partitions in which each interval of the main dimension will be partitioned in Q rectangles. Such a partition is presented in Figure 1(b).

An intuitive heuristic to generate $P \times Q$-way jagged partitions, we call JAG-PQ-HEUR, is to use a 1D partitioning algorithm to partition the main dimension and then partition each interval independently. First, we project the array on the main dimension by summing all the elements along the auxiliary dimension. An optimal 1D partitioning algorithm generates the intervals of the main dimension. Then, for each interval, the elements are projected on the auxiliary dimension by summing the elements along the main dimension. An optimal 1D partitioning algorithm is used to partition each interval. This heuristic have been proposed several times before, for instance in [2].

The algorithm runs in $O((P\log\frac{n_1}{P})^2 + P(Q\log\frac{n_2}{Q})^2)$. Notice that using prefix sum arrays, there is actually no projection to make: the load of interval (i,j) in the main dimension is $L(i,j,1,n_2)$.

We now provide an original analysis of the performance of this heuristic under the hypothesis that all the elements of the load matrix are strictly positive. First, we provide a refinement on the upper bound of the optimal maximum load in the 1D partitioning problem by refining the performance bound of DC (and RB) under this hypothesis.

Lemma 1. *If there is no zero in the array, applying* DirectCut *on a one dimensional array A using m processors leads to a maximum load having the following property:* $L_{max}(DC) \leq \frac{\sum A[i]}{m}(1+\Delta\frac{m}{n})$ *where* $\Delta = \frac{\max_i A[i]}{\min_i A[i]}$.

Proof: The proof is a simple rewriting of the performance bound of DirectCut: $L_{max}(DC) \leq \frac{\sum_i A[i]}{m} + \max_i A[i] \leq \frac{\sum_i A[i]}{m}(1+\Delta\frac{m}{n})$. □

JAG-PQ-HEUR is composed of two calls to an optimal one dimensional algorithm. One can use the performance guarantee of DC to bound the load imbalance at both steps. This is formally expressed in the following theorem.

Theorem 1. *If there is no zero in the array,* JAG-PQ-HEUR *is a $(1+\Delta\frac{P}{n_1})(1+\Delta\frac{Q}{n_2})$-approximation algorithm where $\Delta = \frac{\max_{i,j} A[i][j]}{\min_{i,j} A[i][j]}$, $P < n_1$, $Q < n_2$.*

Proof: Let us first give a bound on the load of the most loaded interval along the main dimension, i.e., the imbalance after the cut in the first dimension. Let C denote the array of the projection of A among one dimension: $C[i] = \sum_j A[i][j]$. We have: $L^*_{max}(C) \leq \frac{\sum_i C[i]}{P}(1+\Delta\frac{P}{n_1})$. Noticing that $\sum_i C[i] = \sum_{i,j} A[i][j]$, we obtain: $L^*_{max}(C) \leq \frac{\sum_{i,j} A[i][j]}{P}(1+\Delta\frac{P}{n_1})$

Let S be the array of the projection of A among the second dimension inside a given interval c of processors: $S[j] = \sum_{i\in c} A[i][j]$. The optimal partition of S respects: $L^*_{max}(S) \leq \frac{\sum_j S[j]}{Q}(1+\Delta\frac{Q}{n_2})$. Since S is given by the partition of C, we have $\sum_j S[j] \leq L^*_{max}(C)$ which leads to $L^*_{max}(S) \leq (1+\Delta\frac{P}{n_1})(1+\Delta\frac{Q}{n_2})\frac{\sum_{i,j} A[i][j]}{PQ}$ □

It remains the question of the choice of P and Q which is solved by the following theorem.

Theorem 2. *The approximation ratio of* JAG-PQ-HEUR *is minimized by $P = \sqrt{m\frac{n_1}{n_2}}$.*

Proof: The approximation ratio of JAG-PQ-HEUR can be written as $f(x) = (1+ax)(1+b/x)$ with $a,b,x > 0$ by setting $a = \frac{\Delta}{n_1}$, $b = \frac{\Delta m}{n_2}$ and $x = P$. The minimum of f is now computed by studying its derivative: $f'(x) = a - b/x^2$. $f'(x) < 0 \iff x < \sqrt{b/a}$ and $f'(x) > 0 \iff x > \sqrt{b/a}$. It implies that f has one minimum given by $f'(x) = 0 \iff x = \sqrt{b/a}$. □

Notice that when $n_1 = n_2$, the approximation ratio is minimized by $P = Q = \sqrt{m}$.

Two algorithms exist to find an optimal $P \times Q$-way jagged partition in polynomial time. The first one has been proposed first by [2] and is constructed by using the 1D algorithm

presented in [9]. The second one, we call JAG-PQ-OPT is a dynamic programming algorithm proposed by [15]. Both algorithms partition the main dimension using a 1D partitioning algorithm using an optimal partition of the auxiliary dimension for the evaluation of the load of an interval.

3.2.2. m-way Jagged Partitions. We introduce the notion of m-way jagged partitions which allows jagged partitions with different numbers of processors in each interval of the main dimension. Indeed, even the optimal partition in the main dimension may have a high load imbalance and allocating more processor to one interval might lead to a better load balance. Such a partition is presented in Figure 1(c). We propose two algorithms to generate m-way jagged partitions. The first one is a heuristic extending the $P\times Q$-way jagged partitioning heuristic. The second one is a polynomial optimal dynamic programming algorithm.

We propose JAG-M-HEUR which is a heuristic similar to JAG-PQ-HEUR. The main dimension is first partitioned in P intervals using an optimal 1D partitioning algorithm. Then each stripe S is allocated a number of processors Q_S which is proportional to the load of the interval. Finally, each interval is partitioned on the auxiliary dimension using Q_S processors by an optimal 1D partitioning algorithm.

Choosing Q_S is a non trivial matter since distributing the processors proportionally to the load may lead to non integral values which might be difficult to round. Therefore, we only distribute proportionally $(m-P)$ processors which allows to round the allocation up: $Q_S = \left\lceil (m-P)\frac{\sum_{i,j\in S} A[i][j]}{\sum_{i,j} A[i][j]} \right\rceil$. Notice that between 0 and P processors remain unallocated. They are allocated, one after the other, to the interval that maximizes $\frac{\sum_{i,j\in S} A[i][j]}{Q_S}$.

An analysis of the performance of JAG-M-HEUR similar to the one proposed for JAG-PQ-HEUR that takes the distribution of the processors into account is now provided.

Theorem 3. *If there is no zero in A,* JAG-M-HEUR *is a $(\frac{m}{m-P}(1+\frac{\Delta}{n_2})+\Delta\frac{m}{Pn_2}(1+\Delta\frac{P}{n_1}))$-approximation algorithm where $\Delta = \frac{\max_{i,j} A[i][j]}{\min_{i,j} A[i][j]}$, $P < n_1$.*

Proof: Let C denote the array of the projection of A among one dimension: $C[i] = \sum_j A[i][j]$. Similarly to the proof of Theorem 1, we have: $L^*_{max}(C) \leq \frac{\sum A[i][j]}{P}(1+\Delta\frac{P}{n_1})$

Let S denote the array of the projection of A among the second dimension inside a given interval c of an optimal partition of C. $S[j] = \sum_{i\in c} A[i][j]$. We have $\sum_j S[j] \leq L^*_{max}(C)$. Then, the number of processors allocated to the stripe is bounded by: $\frac{(m-P)\sum_j S[j]}{\sum_{i,j} A[i][j]} \leq Q_S \leq \frac{(m-P)\sum_j S[j]}{\sum_{i,j} A[i][j]} + 1$. The bound on $\sum_j S[j]$ leads to $Q_S \leq \frac{m-P}{P}(1+\frac{\Delta P}{n_1})+1$.

We now can compute bounds on the optimal partition of stripe S. The bound from Lemma 1 states: $L^*_{max}(S) \leq \frac{\sum_j S[j]}{Q_S}(1+\frac{\Delta Q_S}{n_2})$. The bounds on $\sum_j S[j]$ and Q_S imply $L^*_{max}(S) \leq \frac{\sum A[i][j]}{m}\frac{m}{m-P}(1+\frac{\Delta}{n_2}(\frac{m-P}{P}(1+\frac{\Delta P}{n_1})+1))$.

The load imbalance (and therefore the approximation ratio) is less than $\frac{m}{m-P}(1+\frac{\Delta}{n_2}(\frac{m-P}{P}(1+\Delta\frac{P}{n_1})+1))$, which can be rewritten as $\frac{m}{m-P}(1+\frac{\Delta}{n_2})+\Delta\frac{m}{Pn_2}(1+\Delta\frac{P}{n_1})$. □

This approximation ratio should be compared to the one obtained by JAG-PQ-HEUR: $(1+\Delta\frac{P}{n_1})+\Delta\frac{m}{Pn_2}(1+\Delta\frac{P}{n_1})$. Basically, using m-way partitions trades a factor of $(1+P\frac{\Delta}{n_1})$ to the profit of a factor $\frac{m}{m-P}(1+\frac{\Delta}{n_2})$. Since P should be of the order of $\sqrt{m}$, JAG-M-HEUR should lead to much better performance than JAG-PQ-HEUR for larger m values.

We can also compute the number of stripes P which optimizes the approximation ratio of JAG-M-HEUR.

Theorem 4. *The approximation ratio of* JAG-M-HEUR *is minimized by $P = \frac{m(\sqrt{\Delta(\Delta+n_2)}-\Delta)}{n_2}$.*

Proof: We analyze the function of the approximation ratio in function of the number of stripes: $f(x) = (\frac{m}{m-x}(1+\frac{\Delta}{n_2})+\frac{m\Delta}{xn_2}(1+\frac{\Delta x}{n_1}))$. Its derivative is: $f'(x) = \frac{1+\frac{\Delta}{n_2}}{(m-x)^2} - \frac{\Delta}{n_2x^2}$. The derivative is negative when x tends to 0^+, positive when x tends to $+\infty$ and null when $n_2x^2 + 2m\Delta x - \Delta m^2 = 0$. This equation has a unique positive solution: $x = \frac{m(\sqrt{\Delta(\Delta+n_2)}-\Delta)}{n_2}$ □

This result is fairly interesting. The optimal number of stripes is linear in the number of processors, dependent on Δ and dependent on n_2 but not on n_1. The dependency on Δ makes the determination of P difficult in practice since a few extremal values may have a large impact on the actual Δ without impacting the load balance in practice. Therefore, JAG-M-HEUR will use $\sqrt{m}$ stripes.

We provide another algorithm, JAG-M-OPT which builds an optimal m-way jagged partition in polynomial time using dynamic programming. An optimal solution can be represented by k, the beginning of the last interval on the main dimension, and x, the number of processors allocated to that interval. What remains is a $(m-x)$-way partitioning problem of a matrix of size $(k-1)\times n_2$. It is obvious that the interval $\{(k-1),\ldots,n_1\}$ can be partitioned independently from the remaining array. The dynamic programming formulation is:

$$L_{max}(n_1,m) = \min_{1\leq k\leq n_1, 1\leq x\leq m} \max\ L_{max}(k-1,m-x), 1D(k,n_1,x)$$

where $1D(i,j,k)$ denotes the value of the optimal 1D partition among the auxiliary dimension of the $[i,j]$ interval on k processors.

There are at most n_1m calls to L_{max} to evaluate, and at most n_1^2m calls to $1D$ to evaluate. Evaluating one function call of L_{max} can be done in $O(n_1m)$ and evaluating $1D$ can be done in $O((x\log\frac{n_2}{x})^2)$ using the algorithm from [9].

The algorithm can trivially be implemented in $O((n_1 m)^2 + n_1^2 m^3 (\log \frac{n_2}{m})^2) = O(n_1^2 m^3 (\log \frac{n_2}{m})^2)$ which is polynomial.

However, this complexity is an upper bound and several improvements can be made, allowing to gain up to two orders of magnitude. First of all, the different values of both functions L_{max} and $1D$ can only be computed if needed. Then the parameters k and x can be found using binary search. For a given x, $L_{max}(k-1, m-x)$ is an increasing function of k, and $1D(k, n_1, x)$ is a decreasing function of k. Therefore, their maximum is a bi-monotonic, decreasing first, then increasing function of k, and hence its minimum can be found using a binary search.

Moreover, the function $1D$ is the value of an optimal 1D partition, and we know lower bounds and an upper bound for this function. Therefore, if $L_{max}(k-1, m-x) > UB(1D(k, n_1, x))$, there is no need to evaluate function $1D$ accurately since it does not give the maximum. Similar arguments on lower and upper bound of $L_{max}(k-1, m-x)$ can be used.

Finally, we are interested in building an optimal m-way jagged partition and we use branch-and-bound techniques to speed up the computation. If we already know a solution to that problem, we can use its maximum load l to decide not to explore some of those functions, if the values (or their lower bounds) L_{max} or $1D$ are larger than l.

3.3. Hierarchical Bipartition

Hierarchical bipartitioning techniques consist of obtaining partitions that can be recursively generated by splitting one of the dimensions into two intervals. An example of such a partition is depicted in Figure 1(d). Notice that such partitions can be represented by a binary tree for easy indexing.

A classical algorithm to generate such a partition is Recursive Bisection which has originally been proposed in [21] and that we call in the following `HIER-RB`. It cuts the matrix into two parts of (approximately) equal load and allocates half the processors to each sub-matrix which are partitioned recursively. The dimension being cut in two intervals alternates at each level of the algorithm. This algorithm can be implemented in $O(m \log \max(n_1, n_2))$ since finding the position of the cut can be done using a binary search.

The algorithm was originally designed for a number of processors which is a power of 2 so that the number of processors at each step is even. However, if at a step the number of processors is odd, one part will be allocated $\lfloor \frac{m}{2} \rfloor$ processors and the other part $\lfloor \frac{m}{2} \rfloor + 1$ processors. In such a case, the cutting point is selected so that the load per processor is minimized.

Variants of the algorithm exist based on the decision of the dimension to partition. One variant does not alternate the partitioned dimension at each step but virtually tries both dimensions and selects the one that lead to the best expected load balance [1]. Another variant decides which direction to cut by selecting the direction with longer length.

We propose a polynomial algorithm for generating the optimal hierarchical partition. It uses dynamic programming and relies on the tree representation of a solution of the problem. An optimal hierarchical partition can be represented by the orientation of the cut, the position of the cut (denoted x or y, depending on the orientation), and the number of processors j in the first part.

The algorithm consists in evaluating the function $L_{max}(x_1, x_2, y_1, y_2, m)$ that partitions rectangle (x_1, x_2, y_1, y_2) using m processors.

$$
\begin{aligned}
L_{max}(x_1, x_2, y_1, y_2, m) = \min_j \min \big(& && (1)\\
& \min_x \max(L_{max}(x_1, x, y_1, y_2, j), && (2)\\
& \quad L_{max}(x+1, x_2, y_1, y_2, m-j)), && (3)\\
& \min_y \max(L_{max}(x_1, x_2, y_1, y, j), && (4)\\
& \quad L_{max}(x_1, x_2, y+1, y_2, m-j)) \big) && (5)
\end{aligned}
$$

Equations 2 and 3 consider the partition in the first dimension and Equations 4 and 5 consider it in the second dimension. The dynamic programming provides the position x (or y) to cut and the number of processors (j and $m-j$) to allocate to each part.

This algorithm is polynomial since there are $O(n_1^2 n_2^2 m)$ functions L_{max} to evaluate and each function can naively be evaluated in $O((x_2 - x_1 + y_2 - y1)m)$. Notice that optimization techniques similar to the one used in Section 3.2.2 can be applied. In particular x and y can be computed using a binary search reducing the complexity of the algorithm to $O(n_1^2 n_2^2 m^2 \log(\max(n_1, n_2))))$.

Despite the dynamic programming formulation is polynomial, its complexity is too high to be useful in practice for real sized systems. We extract a heuristic called `HIER-RELAXED`. To partition a rectangle (x_1, x_2, y_1, y_2) on m processors, `HIER-RELAXED` computes the x (or y) and j that optimize the dynamic programming equation, but substitutes the recursive calls to $L_{max}()$ by a heuristic based on the average load: That is to say, instead of making recursive $L_{max}(x, x', y, y', j)$ calls, $\frac{L(x,x',y,y')}{j}$ will be calculated. The values of x (or y) and j provide the position of the cut and the number of processors to allocate to each part respectively. Each part is recursively partitioned. The complexity of this algorithm is $O(m^2 \log(\max(n_1, n_2))))$.

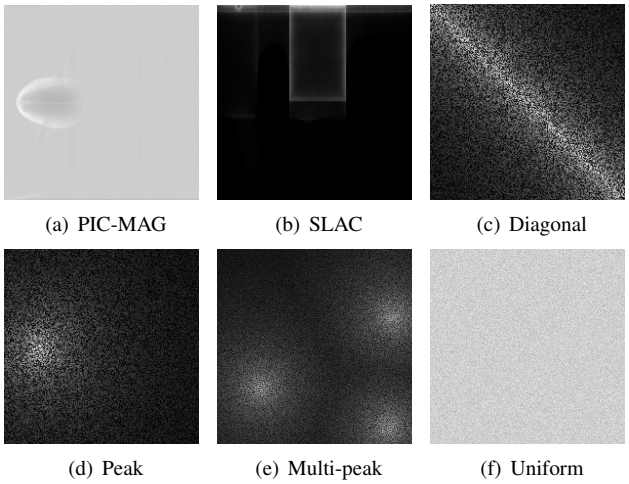
3.4. More General Partitioning Schemes

The considerations on Hierarchical Bipartition can be extended to any kind of recursively defined partitions such as the ones presented in Figures 1(e) and 1(f). As long as there are a polynomial number of possibilities at each level of the

recursion, the optimal partition following this rule can be generated in polynomial time using the dynamic programming technique. The number of functions to evaluate will keep being in $O(n_1^2 n_2^2 m)$. The only difference will be in the cost of evaluating the function calls. In most cases if the pattern is composed of k sections, the evaluation will take $O((\max(n_1, n_2)m)^{k-1})$.

This complexity is too high to be of practical use but it proves that an optimal partition in these classes can be generated in polynomial time. Moreover, those dynamic programming can generate heuristics similarly to HIER-RELAXED.

A natural question is given a maximum load, is it possible to compute an arbitrary rectangular partition? [22] shows that such a problem is NP-Complete and that there is no approximation algorithm of ratio better than $\frac{5}{4}$ unless P=NP. Recent work [23] provides a 2-approximation algorithm which heavily relies on linear programming.



(a) PIC-MAG (b) SLAC (c) Diagonal

(d) Peak (e) Multi-peak (f) Uniform

Figure 2. Examples of real and synthetic instances.

4. Experimental Evaluation

4.1. Experimental Setting

This section presents an experimental study of the some of the presented algorithms. For rectilinear partitions, both the uniform partitioning algorithm RECT-UNIFORM and RECT-NICOL algorithm have been implemented. For jagged partitions, the heuristic and the dynamic programming have been implemented for both $P\times Q$-way and m-way partitions: JAG-PQ-HEUR, JAG-PQ-OPT, JAG-M-HEUR, JAG-M-OPT. Each jagged partitioning algorithm exists in three variants, namely -HOR which considers the first dimension as the main dimension, -VER which considers the second dimension as the main dimension, and -BEST which tries both and selects the one that leads to the best load balance. For hierarchical partitions, both recursive bisection HIER-RB and the heuristic HIER-RELAXED derived from the dynamic programming have been implemented. Each hierarchical bipartition algorithm exists in four variants -LOAD which selects the dimension to partition according to get the best load, -DIST which partitions the longest dimension, and -HOR and -VER which alternate the dimension to partition at each level of the recursion and starting with the first or the second dimension.

The algorithms were tested on the BMI department cluster called Bucki. Each node of the cluster has two 2.4 GHz AMD Opteron(tm) quad-core processors and 32GB of main memory. The nodes run on Linux 2.6.18. The sequential algorithms are implemented in C++. The compiler is g++ 4.1.2 and -O2 optimization was used.

The algorithms are tested on different classes of instances. Some are synthetic and some are extracted from real applications. The first set of instances is called PIC-MAG. These instances are extracted from the execution of a particle-in-cell code which simulates the interaction of the solar wind on the Earth's magnetosphere [6]. In those applications, the computational load of the system is mainly carried by particles. We extracted the distribution of the particles every 500 iterations of the simulations for the first 33,500 iterations. These data are extracted from a 3D simulation. Since the algorithms are written for the 2D case, in the PIC-MAG instances, the number of particles are accumulated among one dimension to get a 2D instance. A PIC-MAG instance at iteration 20,000 can be seen in Figure 2(a). The intensity of a pixel is linearly related to computation load for that pixel (the whiter the more computation). During the course of the simulation, the particles move inside the space leading to values of Δ varying between 1.21 and 1.51.

The SLAC dataset (depicted in Figure 2(b)) is generated from the mesh of a 3D object. Each vertex of the 3D object carries one unit of computation. Different instances can be generated by projecting the mesh on a 2D plane and by changing the granularity of the discretization. This setting match the experimental setting of [9]. In the experiments, we generated instances of size 512x512. Notice that the matrix contains zeroes, therefore Δ is undefined.

Different classes of synthetic squared matrices are also used, these classes are called diagonal, peak, multi-peak and uniform. Uniform matrices (Figure 2(f)) are generated to obtain a given value of Δ: the computation load of each cell is generated uniformly between 1000 and $1000 * \Delta$. In the other three classes, the computation load of a cell is given by generating a number uniformly between 0 and the number of cells in the matrix which is divided by the euclidean distance to a reference point (a 0.1 constant is added to avoid dividing by zero). The choice of the reference point is what makes the difference between the three classes of instances. In diagonal (Figure 2(c)), the reference point is the closest point on the diagonal of the matrix. In peak (Figure 2(d)), the reference point is one point chosen randomly at the beginning of the

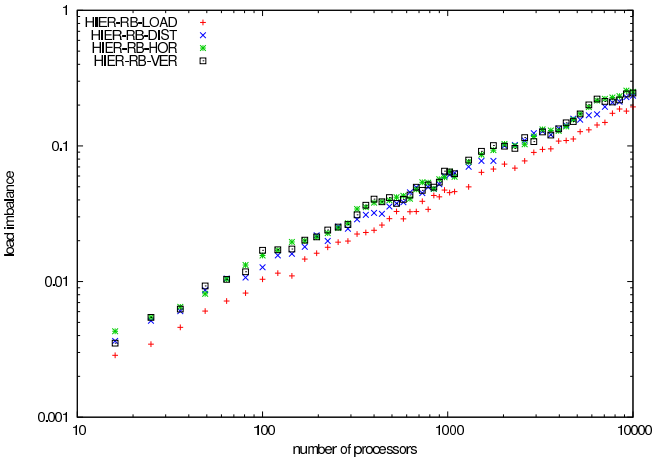


Figure 3. HIER-RB on 1024x1024 Peak.

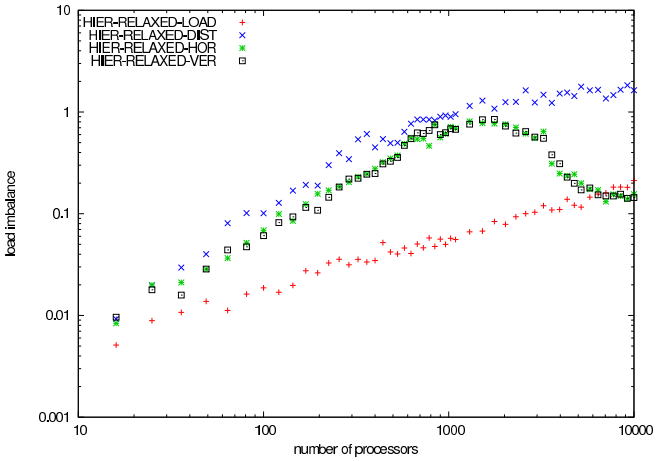


Figure 4. HIER-RELAXED on 512x512 Multi-peak.

execution. In multi-peak (Figure 2(e)), several points (here 3) are randomly generated and the closest one will be the reference point. Those classes are inspired from the synthetic data from [15].

The performance of the algorithms is given using the load imbalance metric defined in Section 2. For synthetic dataset, the load imbalance is computed over 10 instances as follow: $\frac{\sum_I L_{max}(I)}{\sum_I L_{avg}(I)} - 1$. The experiments are run on most square number of processors between 16 and 10,000. Using only square numbers allows us to fix the parameter $P = \sqrt{m}$ for all rectilinear and jagged algorithm.

4.2. Variants of the 2D algorithms

Since there are several variants of the same algorithm, we are first presenting a comparison between the variants of a given algorithm.

Let us start with HIER-RB. There are four variants of this algorithm depending on the dimension that will be partitioned in two. Figure 3 shows the load imbalance of the four variants when the number of processors varies on a 1,024x1,024 instance generated according to the peak rule. In general the load imbalance increases with the number of processors. The HIER-RB-LOAD variant achieves overall the best load balance and, from now on, we will refer to it as HIER-RB. The results obtained on different classes of instances concur with this result and, hence, are omitted.

There are also four variants to the HIER-RELAXED algorithm. Figure 4 shows the load imbalance of the four variants when the number of processors varies on the multi-peak instances of size 512. In general the load imbalance increases with the number of processors for the HIER-RELAXED-LOAD and HIER-RELAXED-DIST. The HIER-RELAXED-LOAD variant achieves overall the best load balance. The load imbalance of the HIER-RELAXED-VER (and HIER-RELAXED-HOR) variant improves past 2,000 processors and seems to converge to the performance of HIER-RELAXED-LOAD. The number of processors where these variants start improving depends on the size of the load matrix. Before convergence, the obtained load balance is comparable to the one obtained by HIER-RELAXED-DIST. The diagonal instances with a size of 4,096 presented in Figure 5 shows this behavior. Since HIER-RELAXED-LOAD leads to the best load imbalance, we will refer to it as HIER-RELAXED.

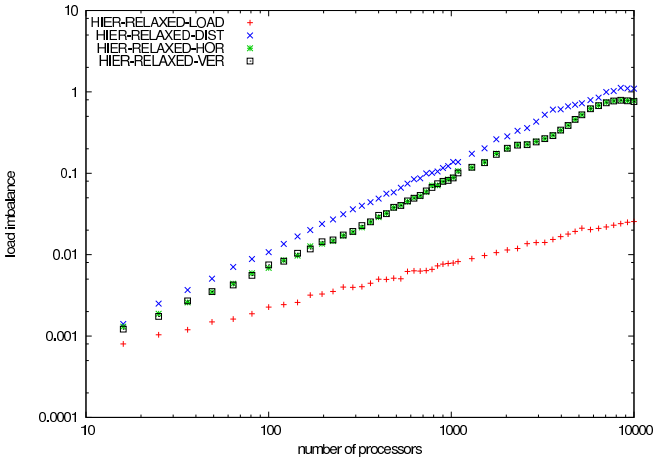


Figure 5. HIER-RELAXED on 4096x4096 Diagonal.

The jagged algorithms have three variants, two depending on whether the main dimension is the first one or the second one and the third tries both of them and takes the best solution. On all the fairly homogeneous instances (i.e., all but the mesh SLAC), the results of the three variants are quite close and the orientation of the jagged partitions does not seem to really matter. However this is not the same in m-way jagged algorithms where the selection of the main dimension can make significant differences on overall load imbalance. Since m-way jagged partitioning is as fast as

heuristic jagged partitioning, trying both dimensions and taking the one with best load imbalance is a good option. From now on, JAG-PQ-HEUR and JAG-M-HEUR will refer to their BEST variant.

4.3. Execution time

In all optimization problems, the trade-off between the quality of a solution and the computation time spent computing it appears. We present in Figure 6 the execution time of the different algorithms on 512x512 Uniform instances with $\Delta = 1.2$ when the number of processors varies. The execution times of the algorithms increase with the number of processors. All the heuristics complete in less than one seconds even on 10,000 processors. The fastest algorithm is obviously RECT-UNIFORM since it outputs trivial partitions. The second fastest algorithm is HIER-RB which computes a partition in 10,000 processors in 18 milliseconds. Then comes the JAG-PQ-HEUR and JAG-M-HEUR heuristics which take about 106 milliseconds to compute a solution of the same number of processors. The running time of RECT-NICOL algorithm is more erratic (probably due to the iterative refinement approach) and it took 448 milliseconds to compute a partition in 10,000 rectangles. The slowest heuristic is HIER-RELAXED which requires 0.95 seconds of computation to compute a solution for 10,000 processors. The computation of JAG-PQ-OPT is much higher, 10 seconds to compute a partition of more than 1,096 parts and 27 seconds for 10,000 parts. The computation time of JAG-M-OPT is not reported on the chart. We never run this algorithm on a large number of processors since it already took 15 minutes to compute a solution for 961 processors. The results on different classes of instances are not reported, but show the same trends. Experiments with larger load matrices show an increase in the execution time of the algorithm. Running the algorithm on matrices of size 8,192x8,192 basically increases the running times by an order of magnitude.

Loading the data and computing the prefix sum array are not included in the presented timing results and take about 40 milliseconds on a 512x512 matrix.

4.4. Jagged Partitioning Schemes

We proposed in Section 3.2.2 a new type of jagged partitioning scheme, namely, m-way jagged, which does not require all the slices of the main dimension to have the same number of processors. This constraint is artificial in most cases and we show that it significantly harms the load balance of an application.

Figure 7 presents the load balance obtained on PIC-MAG at iteration 30,000 with heuristic and optimal $P \times Q$-way jagged algorithms and m-way jagged algorithms. On less than one thousand processors, JAG-M-HEUR,

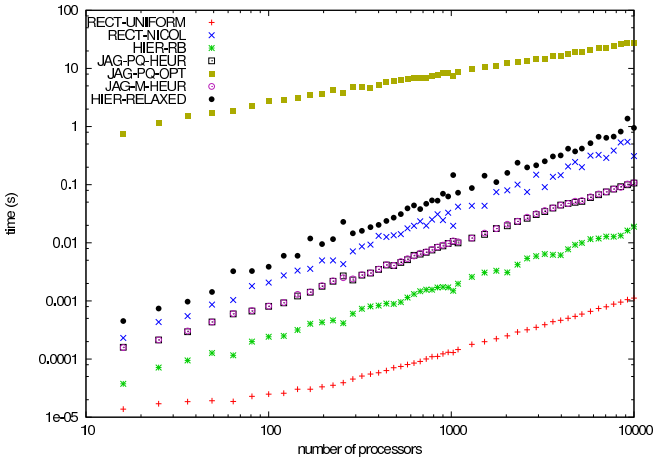


Figure 6. Runtime on 512x512 Uniform with $\Delta = 1.2$.

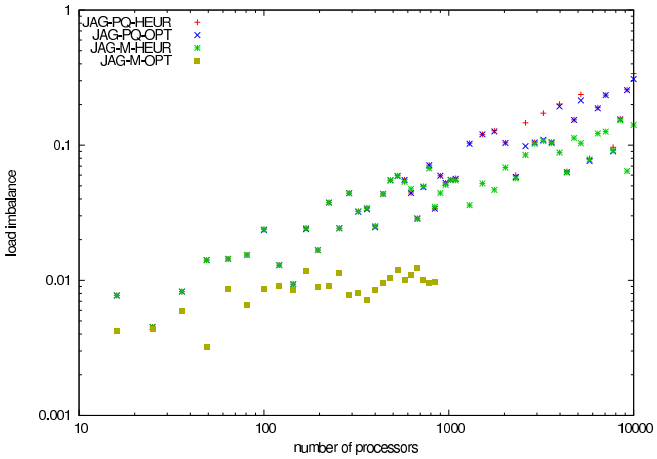


Figure 7. Jagged methods on PIC-MAG iter=30,000.

JAG-PQ-HEUR and JAG-PQ-OPT produce almost the same results (hence the points on the chart are super imposed). Note that, JAG-PQ-HEUR and JAG-PQ-OPT obtain the same load imbalance most of the time even on more than one thousand processors. This indicates that there is almost no room for improvement for the $P \times Q$ heuristic. The second remark is that the m-way jagged heuristic always reaches a better load balance than the $P \times Q$-way jagged partitions.

Figure 8 presents the load imbalance of the algorithms with 6,400 processors for the different iterations of the PIC-MAG application. $P \times Q$ partitions have a load imbalance of 18% while the imbalance of the heuristic m-way partitions varies between 2.5% (at iteration 5,000) and 16% (at iteration 18,000).

In Figure 7, the optimal m-way partition have been computed up to 1,000 processors (on more than 1,000 processors, the runtime of the algorithm becomes prohibitive). It shows an imbalance of about 1% at iteration 30,000 of

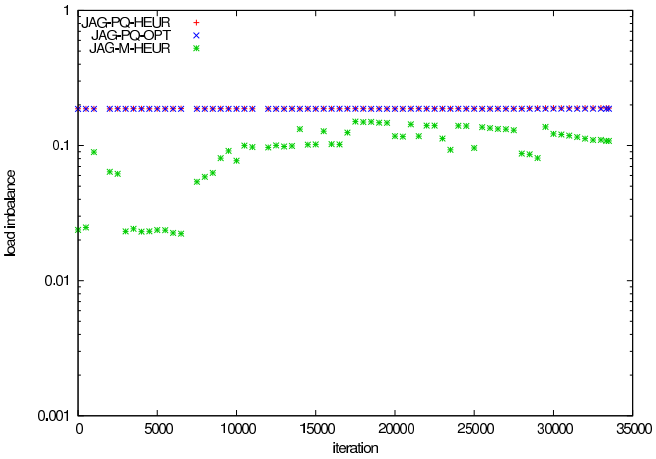


Figure 8. Jagged methods on PIC-MAG with $m = 6400$.

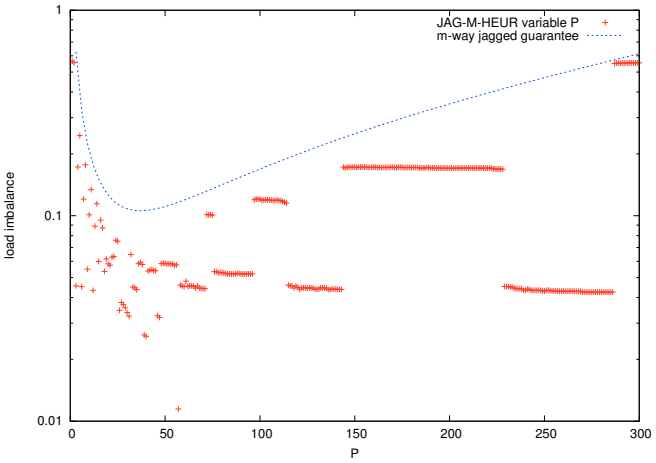


Figure 9. Impact of the number of stripes in JAG-M-HEUR on a 514x514 Uniform instance with $\Delta = 1.2$ and $m = 800$.

the PIC-MAG application on 1,000 processors. This value is much smaller than the 6% imbalance of JAG-M-HEUR. It indicates that there is room for improvement for m-way jagged heuristics. Indeed, the current heuristic uses $\sqrt{m}$ parts in the first dimension, while the optimal is not bounded to that constraint. Figure 9 presents the impact of the number of stripes on the load imbalance of JAG-M-HEUR on a uniform instance as well as the worst case imbalance of the m-way jagged heuristic guaranteed by Theorem 3. It appears clearly that the actual performance follows the same trend as the worst case performance of JAG-M-HEUR. Therefore, ideally, the number of stripes should be chosen according to the guarantee of JAG-M-HEUR. However, the parameters of the formula in Theorem 4 are difficult to estimate accurately and the variation of the load imbalance around that value can not be predicted accurately.

The load imbalance of JAG-PQ-HEUR, JAG-PQ-OPT and JAG-M-HEUR make some waves on Figure 7 when the number of processors varies. Those waves are caused by the imbalance of the partitioning in the main dimension of the jagged partition. Even more, these waves synchronized with the integral value of $\frac{n_1}{\sqrt{m}}$. This behavior is linked to the almost uniformity of the PIC-MAG dataset. The same phenomena induces the steps in Figure 9.

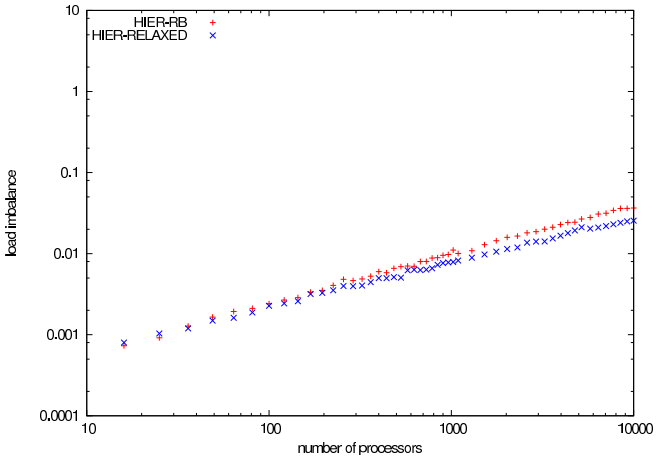


Figure 10. Hierarchical methods on 4096x4096 Diagonal.

4.5. Hierarchical Bipartition

We proposed a new heuristic, HIER-RELAXED, to compute hierarchical bipartitions based on the lesson learned from the dynamic programming formulation for hierarchical bipartition, we proposed in Section 3.3. Notice that we did not implement the dynamic programming algorithm since we expect it to run in hours even on small instances. Figure 10 presents the performance of HIER-RB and HIER-RELAXED on the diagonal matrix instances of size 4,096. It is clear that HIER-RELAXED leads to a better load balance than HIER-RB. However, the performance of HIER-RELAXED might be very erratic when the instance changes slightly. For instance, on Figure 11 the performance of HIER-RELAXED during the execution of the PIC-MAG application is highly unstable.

4.6. Which algorithm to choose ?

The main question remains. Which algorithm should be chosen to optimize an application's performance ?

From the algorithm we presented, we showed that m-way jagged partitions provides a better solution than an optimal $P \times Q$-way jagged partition. It is therefore better than rectilinear partitions as well. The computation of an optimal m-way jagged partition is too slow to be used in a real system. It remains to decide between JAG-M-HEUR, HIER-RB and HIER-RELAXED.

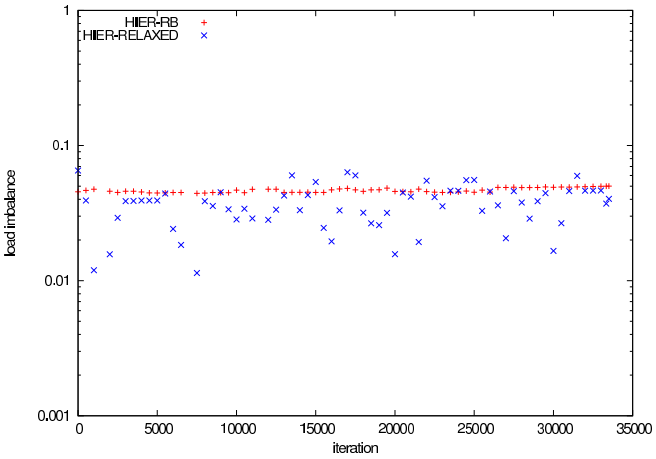


Figure 11. Hierarchical methods on PIC-MAG with $m = 400$.

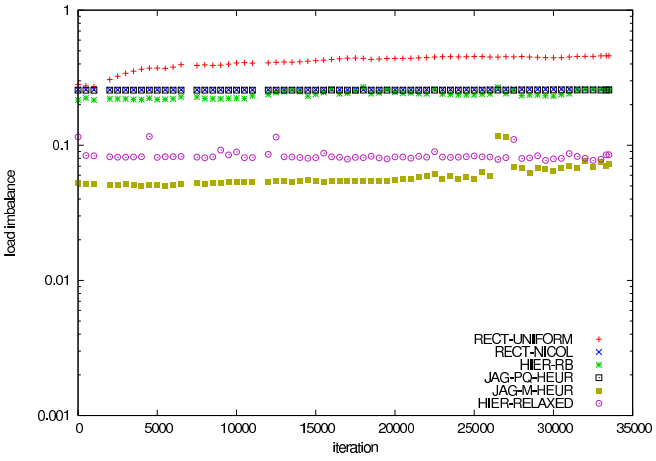


Figure 12. All heuristics on PIC-MAG with $m = 9216$.

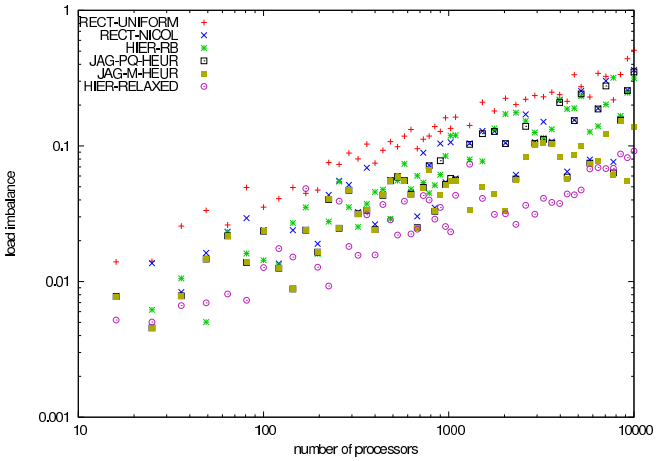


Figure 13. All heuristics on PIC-MAG iter=20,000.

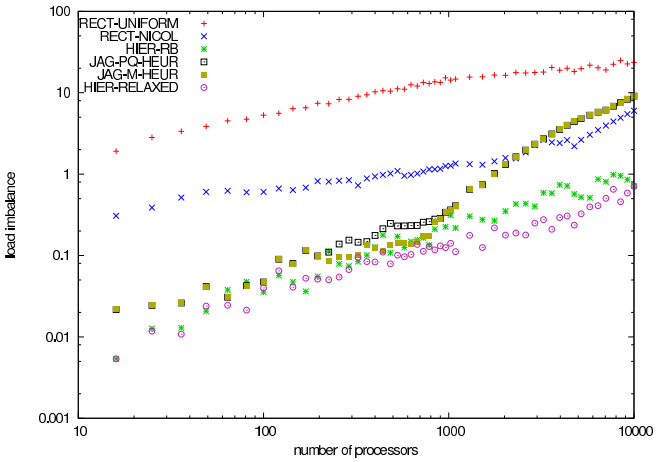


Figure 14. All heuristics on SLAC.

Figure 12 shows the performance of the PIC-MAG application on 9,216 processors. The RECT-UNIFORM partitioning algorithm is given as a reference. It achieves a load imbalance that grows from 30% to 45%. Both RECT-NICOL and JAG-PQ-HEUR reach a constant 28% imbalance over time. HIER-RB is usually slightly better and achieves a load imbalance that varies between 20% and 30%. HIER-RELAXED achieves most of the time a much better load imbalance, rarely over 10% and typically between 8% and 9%. JAG-M-HEUR achieves the best performance in all iterations (but two) of all the tested algorithms by keeping the load imbalance between 5% and 8%.

Figure 13 shows the performance of the algorithms while varying the number of processors at iteration 20,000. The conclusions on RECT-UNIFORM, RECT-NICOL, JAG-PQ-HEUR and HIER-RB stand. Depending on the number of processors, the performance of JAG-M-HEUR varies and in general HIER-RELAXED leads to the best performance, in this test.

Figure 14 presents the performance of the algorithms on the mesh based instance SLAC. Due to the sparsity of the instance, most algorithms get a high load imbalance. Only the hierarchical partitioning algorithms manage to keep the imbalance low and HIER-RELAXED gets a lower imbalance than HIER-RB.

The results indicate that as it stands, the algorithms HIER-RELAXED and JAG-M-HEUR, we proposed, are the one to choose to get a good load balance. However, we believe a developer should be cautious when using HIER-RELAXED because of the erratic behavior it showed in some experiments (see Figure 11) and because of its not-that-low running time (up to one second on 10,000 processors according to Figure 6). JAG-M-HEUR seems much a more stable heuristic. The bad load balance it presents on Figure 13 is due to a badly chosen number of partitions in the first dimension.

5. Conclusion

Partitioning spatially localized computations evenly among processors is a key step in obtaining good performance in a large class of parallel applications. In this work, we focused on partitioning a matrix of positive integer using rectangular partitions to obtain a good load balance. We introduced the new class of solutions called m-way jagged partitions, designed polynomial optimal algorithms and heuristics for m-way partitions. Using theoretical worst case performance analyses and simulations based on logs of two real applications and synthetic data, we showed that the `JAG-M-HEUR` and `HIER-RELAXED` heuristics we proposed get significantly better load balances than existing algorithms.

As a future work, we plan to investigate the effect of these different partitioning schemes in communication cost, as well as taking into account data migration costs in dynamic applications. We are also planning to integrate the proposed algorithms in a real dynamic application and study their end-to-end effects.

Acknowledgment

We thank to Y. Omelchenko and H. Karimabadi for providing us with the PIC-MAG data; and R. Lee, M. Shephard, and X. Luo for the SLAC data.

References

[1] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[2] A. Pinar and C. Aykanat, "Sparse matrix decomposition with optimal load balancing," in *Proc. of HiPC 1997*, 1997.

[3] M. Ujaldon, S. Sharma, E. Zapata, and J. Saltz, "Experimental evaluation of efficient sparse matrix distributions," in *proc. of SuperComputing'96*, 1996.

[4] H. Kutluca, T. Kurc, and C. Aykanat, "Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids," *J. of Supercomputing*, vol. 15, pp. 51–93, 2000.

[5] S. J. Plimpton, D. B. Seidel, M. F. Pasik, R. S. Coats, and G. R. Montry, "A load-balancing algorithm for a parallel electromagnetic particle-in-cell code," *Computer Physics Communications*, vol. 152, no. 3, pp. 227 – 241, 2003.

[6] H. Karimabadi, H. X. Vu, D. Krauss-Varban, and Y. Omelchenko, "Global hybrid simulations of the earths magnetosphere," *Numerical Modeling of Space Plasma Flows*, vol. 359 of Astronomical Society of the Pacific Conference Series, Dec. 2006.

[7] A. L. Lastovetsky and J. J. Dongarra, "Distribution of computations with constant performance models of heterogeneous processors," in *High Performance Heterogeneous Computing*. John Wiley & Sons, 2009, ch. 3.

[8] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, pp. 974–996, 2004.

[9] D. Nicol, "Rectilinear partitioning of irregular data parallel computations," *Journal of Parallel and Distributed Computing*, vol. 23, pp. 119–134, 1994.

[10] Y. Han, B. Narahari, and H.-A. Choi, "Mapping a chain task to chained processors," *Information Processing Letter*, vol. 44, pp. 141–148, 1992.

[11] F. Manne and B. Olstad, "Efficient partitioning of sequences," *IEEE Transactions on Computers*, vol. 44, no. 11, pp. 1322–1326, 1995.

[12] S. Miguet and J.-M. Pierson, "Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing," in *Proc. of HPCN Europe '97*, 1997, pp. 550–564.

[13] G. N. Frederickson, "Optimal algorithms for partitioning trees and locating p-centers in trees," Purdue University, Tech. Rep. CSD-TR-1029, 1990, revised 1992.

[14] B. Aspvall, M. M. Halldórsson, and F. Manne, "Approximations for the general block distribution of a matrix," *Theor. Comput. Sci.*, vol. 262, no. 1-2, pp. 145–160, 2001.

[15] F. Manne and T. Sørevik, "Partitioning an array onto a mesh of processors," in *Proc of PARA '96*, 1996, pp. 467–477.

[16] H. P. F. Forum, "High performance FORTRAN language specification, version 2.0," CRPC, Tech. Rep. CRPC-TR92225, Jan. 1997.

[17] M. Grigni and F. Manne, "On the complexity of the generalized block distribution," in *Proc. of IRREGULAR '96*, 1996, pp. 319–326.

[18] S. Khanna, S. Muthukrishnan, and S. Skiena, "Efficient array partitioning," in *Proc. of ICALP '97*, 1997, pp. 616–626.

[19] D. R. Gaur, T. Ibaraki, and R. Krishnamurti, "Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem," *J. Algorithms*, vol. 43, no. 1, pp. 138–152, 2002.

[20] S. Muthukrishnan and T. Suel, "Approximation algorithms for array partitioning problems," *Journal of Algorithms*, vol. 54, pp. 85–104, 2005.

[21] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transaction on Computers*, vol. C36, no. 5, pp. 570–580, 1987.

[22] S. Khanna, S. Muthukrishnan, and M. Paterson, "On approximating rectangle tiling and packaging," in *proceedings of the 19th SODA*, 1998, pp. 384–393.

[23] K. Paluch, "A new approximation algorithm for multidimensional rectangle tiling," in *Proceedings of ISAAC*, 2006.