LOAD-BALANCING SPATIALLY LOCATED COMPUTATIONS USING RECTANGULAR PARTITIONS

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of the Ohio State University

By

Erdeniz Ö. Baş,

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Master's Examination Committee:

Ümit V. Çatalyürek, Advisor

Radu Teodorescu

ABSTRACT

Distributing spatially located heterogeneous workloads is an important problem in parallel scientific computing. Particle-in-cell simulators, ray tracing and partial differential equations are some of the applications with spatially located workload. We investigate the problem of partitioning such workloads (represented as a matrix of non-negative 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: rectilinear, jagged and hierarchical. 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. Balancing the load does not guarantee to minimize the total runtime of an application. In order to achieve that, one must also take into account the communication cost. Rectangle shaped partitioning inherently keeps communications small, yet one should proactively minimize them. The algorithms we propose are tested in simulation on a wide set of instances and compared to state of the art algorithm. Results show that m-way jagged partitions are low in total communication cost and practical to use.

To my mother...

ACKNOWLEDGMENTS

I would like to express my appreciation to my parents Demet and Mehmet for their continuous support. Without their devotions, I wouldn't be able to complete my education.

During my MS studies, I had the opportunity to work with Prof. Umit Catalyurek, one of the best advisors a grad student can have. He created a welcoming environment that allowed me to focus on my research without distractions. With his helpfulness and knowledge, I was able to put more value on my thesis. I would like to thank him for all his efforts.

Lastly, I would like to thank Erik Saule who answered all my questions promptly and explained complex concepts tirelessly. He was not only a teacher but also a friend who kept my morale high. I learned quite a lot from him and this thesis is shaped by his valuable thoughts.

VITA

1986	Born in Kirsehir, Turkey
2009	B.E. in Computer Engineering, Ege University, Turkey
2010-Present	Graduate Research Associate, The Ohio State University

PUBLICATIONS

E. Saule, E. Ö. Baş, and Ümit V. Çatalyürek. Partitioning spatially located computations using rectangles. In Proc. of 25th IEEE International Parallel and Distributed Processing Symposium, 2011.

E. Saule, E. Ö. Baş, and Ümit V. Çatalyürek. Load-balancing spatially located computations using rectangular partitions. Technical Report arXiv:1104.2566v1, ArXiv, Apr 2011.

FIELDS OF STUDY

Major Field: Computer Science

Specialization: High Performance Computing

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

CHAPTER 1

INTRODUCTION

1.1 Motivation

To achieve good efficiency when using a parallel platform, one must distribute the computations and the required data to the processors of the parallel machine. If the computation tasks are independent, their parallel processing falls in the category of pleasantly parallel tasks. Even in such cases, when computation time of the tasks are not equal, obtaining the optimal load balance to achieve optimum execution time becomes computationally hard and heuristic solutions are used [26]. Furthermore, most of the time some dependencies exist between the tasks and some data must be shared or exchanged frequently, making the problem even more complicated.

A large class of application see its computations take place in a geometrical space of typically two or three dimensions. Different types of applications fall into that class. Particle-in-cell (PIC) simulation [19, 37] is an implementation of the classical mean-field approximation of the many-body problem in physics. Typically, thousands to millions of particles are located in cells, which are a discretization of a field. The application iteratively updates the value of the field in a cell, based on the state of the particles it contains and the value of the neighboring cells, and then the state of the particles, based on its own state and the state of the cell it belongs to. Direct volume rendering [24] is an application that use rendering algorithm similar to raycasting in

a scene of semi-transparent objects without reflection. For each pixel of the screen, a ray orthogonal to the screen is cast from the pixel and color information will be accumulated over the different objects the ray encounters. Each pixel therefore requires an amount of computation linear to the number of objects crossed by the ray and neighboring pixels are likely to cross the same objects. Partial Differential Equation can be computed using mesh-based computation. For instance [18] solves heat equation on a surface by building a regular mesh out of it. The state of each node of the mesh is iteratively updated depending on the state of neighboring nodes. For load balancing purpose, [31] maps the mesh to a discretized two dimensional space. Another application can be found in 3D engines where the state of the world is iteratively updated and where the updates on each object depends on neighboring objects (for instance, for collision purpose) [1]. Linear algebra operations can potentially also benefit from such techniques [34, 40, 41].

1.2 Focus

In this work, our goal is to balance the load of such applications while keeping communication low. In the literature, load balancing techniques can be broadly divided into two categories: geometric and connectivity-based. Geometric methods (such as [5, 33]) leverages the fact that computations which are close by in the space are more likely to share data than computations that are far in the space, by dividing the load using geometric properties of the workload. Methods from that class often rely on a recursive decomposition of the domain such as octrees [10] or they rely on space filling curves and surfaces [2, 3]. Connectivity-based methods usually model the load balancing problem through a graph or an hypergraph weighted with computation volumes on the nodes and communication volumes on the edges or hyper edges (see for instance [9, 39]). Connectivity-based techniques lead to good partitions

but are usually computationally expensive and require to build an accurate graph (or hypergraph) model of the computation. They are particularly well-suited when the interactions between tasks are irregular. Graphs are useful when modeling interactions that are exactly between two tasks, and hypergraph are useful when modeling more complex interactions that could involve more than two tasks [7, 17].

When the interactions are regular (structured) one can use methods that takes the structure into account. For example, when coordinate information for tasks are available, one can use geometric methods which leads to "fast" and effective partitioning techniques. In geometric partitioning, one prefers to partition the problem into connex and compact parts so as to minimize communication volumes. Rectangles (and rectangular volumes) are the most preferred shape because they implicitly minimize communication, do not restrict the set of possible allocations drastically, are easily expressed and allow to quickly find which rectangle a coordinate belongs to using simple data structures. Hence, in this work, we will only focus partitioning into rectangles.

In more concrete terms, this thesis primarily addresses the problem of partitioning a two-dimensional load matrix composed of non-negative numbers into a given number of rectangles (processors) so as to minimize the load of the most loaded rectangle; the most loaded rectangle is the one whose sum of the element it contains is maximal. The problem is formulated so that each element of the array represents a task and each rectangle represents a processor. Computing the optimal solution for this problem has been shown to be NP-Hard [15]. Therefore, we focus on algorithms with low polynomial complexity that lead to good solutions.

To obtain good performance one also needs to take into account communications. A good algorithm has to achieve a low communication cost within a given communication model. A small sacrifice from load imbalance is acceptable if that will gain

significant reduce in communication. Ultimately, one has to reduce execution time which is a complex function of both communication and computation pattern. One of the functions that model the total runtime is given in [9], which builds on the hypergraph model presented in [7]. (A similar formulation that leverages graph model also exist [38].) The hypergraph model presented in [7] reduces minimizing total volume in sparse matrix-vector multiplication into K-way graph partitioning problem. Even though there are tools for graph [20] and hypergraph [8, 21] partitioning algorithms, the algorithms used in those tools are inherently computationally expensive.

Similar to [9, 38], we consider two types of communication. The first one is occurs during normal execution of the application. The second one is for rebalancing of the load after it changes due to dynamic nature of the application. The other objective of this thesis is to give communication cost comparison of those algorithms based on six metrics that represent communication cost best.

Several previous work tackles a similar problem but they usually presents only algorithms from one class with no experimental validation or a very simple one. These works are referenced in the text when describing the algorithm they introduce. Kutluca et al. [24] is the closest related work. They are tackling the parallelization of a Direct Volume Rendering application whose load balancing is done using a very similar model. They survey rectangle based partition but also more general partition generated from hypergraph modeling and space filling curves. The experimental validation they propose is based on the actual runtime of the Direct Volume Rendering application.

The approach we are pursuing in this work is to consider different classes of rectangular partitioning. Simpler structures are expected to yield bad load balance but to be computed quickly while more complex structures are expected to give good load balance but lead to higher computation time. For each class, we look for optimal

algorithms and heuristics. Several algorithms to deal with this particular problem which have been proposed in the literature are described and analyzed. One original class of solution is proposed and original algorithms are presented and analyzed.

The theoretical analysis of the algorithms is accompanied by an extensive experimentation evaluation of the algorithms to decide which one should be used in practice. The experimentation is composed of various randomly generated datasets and two datasets extracted from two applications, one following the particle-in-cell paradigm and one following the mesh-based computation paradigm.

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

- Algorithmic engineering techniques are used to create hybrid partitioning scheme that provides slower algorithms but with higher quality.

- Important two-dimensional algorithms are analyzed in terms of communication cost in a given partition (application cost)
- Important two-dimensional algorithms are analyzed in terms of rebalancing communication cost (migration cost)
- Implementation details of are provided

Similar classes of solutions are used in the problem of partitioning an equally loaded tasks onto heterogeneous processors (see [25] for a survey). This is a very different problem which 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 thesis is organized as follows. Chapter 2 presents new and known load balancing algorithms. The algorithms are evaluated in on synthetic dataset as well as on dataset extracted from two real simulation codes. Chapter 3 analyzes two-dimensional partitioning algorithms in terms of communication cost. Chapter 4 discusses implementation details and interesting design decisions. Conclusive remarks are given in Chapter 5.

CHAPTER 2

LOAD BALANCING ALGORITHMS

2.1 Model and Preliminaries

2.1.1 Problem Definition

Let A be a two dimensional array of $n_1 \times n_2$ non-negative 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 thesis we are only interested in rectangular allocations, and we will use 'rectangle' and 'processor' interchangeably. That is to say, a solution is a set R 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 R is $L_{max} = \max_{r_i} L(r_i)$. We will denote by L^*_{max} the minimal cost 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 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 $\rho - 1$.

2.1.2 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. An extensive theoretical and experimental comparison of those 1D algorithms has been given in [35]. In [35], the fastest optimal 1D partitioning algorithm is `NicolPlus`; it is an algorithmically engineered modification of [31], which uses a subroutine proposed in [16]. A slower optimal algorithm using dynamic programming was proposed in [27]. Different heuristics have also been developed [29, 35]. Frederickson [12] 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 thesis `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 [29]) 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 [16]. 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: $L^*_{max} \leq \frac{\sum_i A[i]}{m} + \max_i A[i]$.

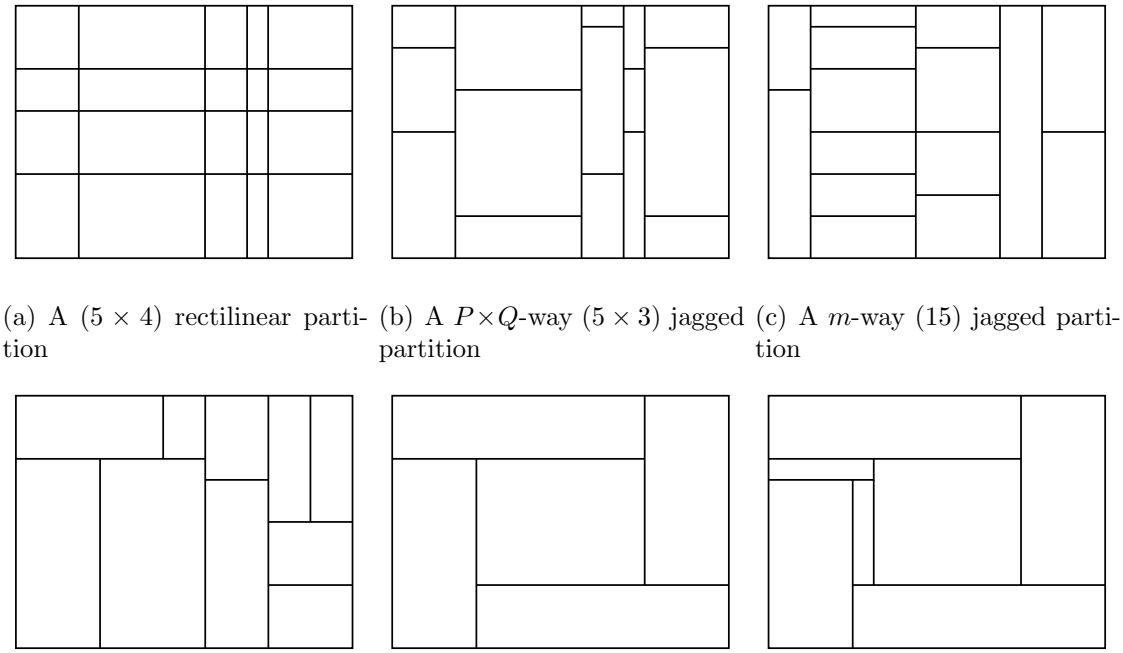
A widely known heuristic is `Recursive Bisection (RB)` which recursively splits the array into two parts of similar load and allocates half the processors to each part. This algorithm leads to a solution such that $L_{max}(RB) \leq \frac{\sum_i A[i]}{m} + \max_i A[i]$ and therefore is a 2-approximation algorithm [35]. It has a runtime complexity of $O(m \log n)$.

The optimal solution can be computed using dynamic programming [27]. 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 [31] relies on the parametric search algorithm proposed in [16]. 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 $O(m \log n)$. [16] 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 [31] 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 a binary search, and the array slicing technique of [16] can be used to reach a complexity of $O((m \log \frac{n}{m})^2)$. Recent work [35] 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.



(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 bipartition

(e) A spiral partition

(f) Another partition

Figure 2.1: Different structures of partitions.

2.2 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. Samples of the considered structures are presented in Figure 2.1. Each structure is a generalization of the previous one.

Table 2.1 summarizes the different algorithms discussed in this thesis. Their worst-case complexity and theoretical guarantees are given.

Name	Parameters	Worst Case Complexity	Theoretical Guarantee
`RECT-UNIFORM`	P,Q	$O(PQ)$	-
`RECT-NICOL`[31]	P,Q	$O(n_1 n_2(Q(P\log\frac{n_1}{P})^2 + P(Q\log\frac{n_2}{Q})^2))$	Better than `RECT-UNIFORM`.
`JAG-PQ-HEUR`[40, 34]	P,Q	$O((P\log\frac{n_1}{P})^2 + P(Q\log\frac{n_2}{Q})^2)$	$[\star]LI \leq (1+\Delta\frac{P}{n_1})(1+\Delta\frac{Q}{n_2}) - 1$
`JAG-PQ-OPT`[28, 34]	P,Q	$O((PQ\log\frac{n_1}{P}\log\frac{n_2}{Q})^2)$ or $O(n_1\log n_1(P + (Q\log\frac{n_2}{Q})^2))$	Optimal for $P\times Q$-way jagged partitioning. Better than all of the above.
`JAG-M-HEUR`[$\star$]	P	$O((P\log\frac{n_1}{P})^2 + (m\log\frac{n_2}{m})^2)$	$LI \leq (\frac{m}{m-P} + \frac{m\Delta}{Pn_2} + \frac{\Delta^2 m}{n_1 n_2}) - 1$
`JAG-M-PROBE`[$\star$]	P dividers	$O((m\log\frac{n_2 P}{m})^2)$	Optimal for the parameters.
`JAG-M-ALLOC`[$\star$]	P, Q_s	$O((P\log\frac{n_1}{P}\max_S Q_S\log\frac{n_2}{Q_S})^2)$	Optimal for the parameters.
`JAG-M-HEUR-PROBE`[$\star$]	P	$O((P\log\frac{n_1}{P})^2 + (m\log\frac{n_2 P}{m})^2)$	Better than `JAG-M-HEUR`.
`JAG-M-OPT`[$\star$]	-	$O(n_1^2 m^3(\log\frac{n_2}{m})^2)$	Optimal for m-way jagged partitioning. Better than all of the above.
`HIER-RB`[5]	-	$O(m\log\max(n_1, n_2))$	-
`HIER-RELAXED`[$\star$]	-	$O(m^2\log\max(n_1, n_2))$	-
`HIER-OPT`[$\star$]	-	$O(n_1^2 n_2^2 m^2\log\max(n_1, n_2))$	Optimal for hierarchical bisection. Better than all of the above.

Table 2.1: Summary of the presented algorithms. Algorithms and results introduced in this thesis are denoted by a [$\star$]. LI stands for Load Imbalance. $\Delta = \frac{\max_{i,j} A[i][j]}{\min_{i,j} A[i][j]}$, if $\forall i, j, A[i][j] > 0$.

2.3 Rectilinear Partitions

Rectilinear partitions (also called General Block Distribution in [4, 28]) organize the space according to a $P \times Q$ grid as shown in Figure 2.1(a). This type of partitions is often used to optimize communication and indexing and has been integrated in the High Performance Fortran standard [11]. 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 naïve partition that balances the area and not the load.

[15] implies that computing the optimal rectilinear partition is an NP-Hard problem. [4] points out that the NP-completeness proof in [15] also implies that there is no $(2-\epsilon)$-approximation algorithm unless P=NP. We can also remark that the proof is valid for given values of P and Q, 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.

[31] (and [28] independently) proposed an iterative refinement heuristic algorithm that we call `RECT-NICOL` in the remaining of this thesis. 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. The algorithm runs until the last 2 iterations return the same partitions. 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 [31] 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 514x514 matrix up to 10,000 processors). [4] shows it is a $\theta(\sqrt{m})$-approximation when $P = Q = \sqrt{m}$.

The first constant approximation algorithm for rectilinear partitions has been proposed by [23] but neither the constant nor the actual complexity is given. [4] claims it is a 120-approximation that runs in $O(n_1 n_2)$.

[4] 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 $O(n(\sqrt{m}\log n)^2)$. [4] claims that despite the approximation ratio is not constant, it is better in practice than the algorithm proposed in [23].

[13] 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.

[30] 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 $O((n_1 + n_2 + PQ)P\log(n_1 n_2))$.

2.3.1 Jagged Partitions

Jagged partitions (also called Semi Generalized Block Distribution in [28]) 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 2.1(b) and 2.1(c). The layout

of jagged partitions also allows to easily locate which rectangle contains a given element [40].

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

$P \times Q$-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 2.1(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 [40, 34].

The algorithm runs in $O((P \log \frac{n_1}{P})^2 + P(Q \log \frac{n_2}{Q})^2)$. Prefix sum arrays avoid redundant projections: the load of interval (i, j) in the main dimension can be simply computed as $L(i, j, 1, n_2)$.

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, we call `JAG-PQ-OPT-NICOL`, has been proposed first by [34] and is constructed by using the 1D algorithm presented in [31]. This algorithm is of complexity $O((PQ \log \frac{n_1}{P} \log \frac{n_2}{Q})^2)$. The second one, we call `JAG-PQ-OPT-DP` is a dynamic programming algorithm proposed by [28]. 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. The complexity of `JAG-PQ-OPT-DP` is $O(n_1 \log n_1 (P + (Q \log \frac{n_2}{Q})^2))$.

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 2.1(c). We propose four algorithms to generate m-way jagged partitions. The first one is `JAG-M-HEUR`, a heuristic extending the $P \times Q$-way jagged partitioning heuristic. The second algorithm generates the optimal m-way jagged partition for given intervals in the main dimension, leading to `JAG-M-HEUR-PROBE`. Then, the third algorithm, called `JAG-M-ALLOC`, generates the optimal m-way jagged partition for a given number of interval provided the number of

processor inside each interval is known. Finally, we present `JAG-M-OPT`, 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 which define P stripes. 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* $(\frac{m}{m-P} + \frac{m\Delta}{Pn_2} + \frac{\Delta^2 m}{n_1 n_2})$*-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 $L^*_{max}(S) \leq \frac{\sum A[i][j]}{m}(\frac{m}{m-P} + \frac{m}{P}\frac{\Delta}{n_2} + \frac{\Delta^2 m}{n_1 n_2})$.

The load imbalance (and therefore the approximation ratio) is less than $(\frac{m}{m-P} + \frac{m}{P}\frac{\Delta}{n_2} + \frac{\Delta^2 m}{n_1 n_2})$. □

This approximation ratio should be compared to the one obtained by `JAG-PQ-HEUR` which can be rewritten as $((1+\Delta\frac{P}{n_1}) + \frac{\Delta m}{P n_2} + \frac{\Delta^2 m}{n_1 n_2})$. Basically, using m-way partitions trades a factor of $(1+\frac{P\Delta}{n_1})$ to the profit of a factor $\frac{m}{m-P}$.

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* $P = \frac{\sqrt{\Delta^2(m^2-1) - n_2} - \Delta m}{n_2 - \Delta}$.

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

This result is fairly interesting. The optimal number of stripes depends of Δ and depends of n_2 but not of n_1. The dependency of Δ makes the determination of P difficult in practice since a few extremal values may have a large impact on the computed P without improving the load balance in practice. Therefore, `JAG-M-HEUR` will

use $\sqrt{m}$ stripes. The complexity of `JAG-M-HEUR` is $O((P \log \frac{n_1}{P})^2 + \sum_S (Q_S \log \frac{n_2}{Q_S})^2)$ which in the worst case is $O((P \log \frac{n_1}{P})^2 + (m \log \frac{n_2}{m})^2)$

We now explain how one can build the optimal jagged partition provided the partition in the main dimension is given. This problem reduces to partitioning P one dimensional arrays using m processors in total to minimize L_{max}. [16] states that the proposed algorithms apply in the presence of multiple chains but does not provide much detail. We explain how to extend `NicolPlus` [35] to the case of multiple one dimensional arrays.

We now first explain the algorithm `PROBE-M` for partitioning multiple arrays that test the feasibility of a given maximum load L_{max}.

The main idea behind `PROBE-M` is to compute for each one dimensional array how many processors are required to achieve a maximum load of L_{max}. For one array, the number of processors require to achieve a load of L_{max} is obtained by greedily allocating intervals maximal by inclusion of load less than L_{max}. The boundary of these intervals can be found in $O(\log n)$ by a binary search. Across all the arrays, there is no need to compute the boundaries of more than m intervals, leading to an algorithm of complexity $O(m \log n)$.

[16] reduces the complexity of the one dimensional partitioning problem to $O(m \log \frac{n}{m})$ by slicing the array in m chunks. That way, one has first to determine in which chunk the borders of the intervals are, and then perform a binary search in the chunk. Provided there are m intervals to generate, the cost of selecting the right chunk is amortized. But it does not directly apply to the multiple array partitioning problem. Indeed, slicing the array in such a manner will lead to a complexity of $O(m \log \frac{n}{m} + Pm)$. However, slicing the arrays in chunk of size $\frac{nP}{m}$ leads to having at most $m+P$ chunks. Therefore, `PROBE-M` has a complexity of $O(m \log \frac{nP}{m} + m + P) = O(m \log \frac{nP}{m})$.

Notice that the engineering presented in [35] for the single array case that use an

upper bound and a lower bound on the position of each boundary can still be used when there are multiple arrays with `PROBE-M`. When the values of L_{max} decreases, the ith cut inside one array is only going to move toward the beginning of the array. Conversely, when L_{max} increase, the ith cut inside one array is only going to move toward the end of the array. One should notice that the number of processors allocated to one array might vary when L_{max} varies.

With `PROBE-M`, one can solve the multiple array partitioning problem in multiple way. An immediate one is to perform a binary search on the values of L_{max}. It is also possible to reuse the idea of `NicolPlus` which rely on the principle that the first interval is either maximal such that the load is an infeasible maximum load or minimal such that the load is a feasible maximum load. The same idea applies by taking the intervals of each array in the same order `PROBE-M` considers them. The windowing trick still applies and leads to an algorithm of complexity $O((m \log \frac{nP}{m})^2)$. Given the stripes in the main dimension, `JAG-M-PROBE` is the algorithm that applies the modified version of `NicolPlus` to generate an m-way partition.

Other previous algorithms apply to this problem. For instance, [6] solves the multiple chains problem on host-satellite systems. One could certainly use this algorithm but the runtime complexity is $O(n^3 m \log n)$. Another way to solve the problem can certainly be derived from the work of Frederickson [12].

`JAG-M-HEUR-PROBE` is the algorithm that uses the stripes obtained by `JAG-M-HEUR` and then applies `JAG-M-PROBE`.

Given a number of stripes P and the number of processors Q_S inside each stripe, one can compute the optimal m-way jagged partition. The technique is similar to the optimal $P \times Q$-way jagged partitioning technique shown in [34]. `NicolPlus` gives an optimal partition not only on one dimensional array but on any one dimension structure where the load of intervals are monotonically increasing by inclusion. When

`NicolPlus` needs the load of an interval, one can return the load of the optimal Q_S-way partition of the auxiliary dimension, computed in $O((Q_S \log \frac{n_2}{Q_S})^2)$.

To generate the m-way partition, one needs to modify `NicolPlus` to keep track of which stripe an interval represents to return the load of the optimal partition of the auxiliary dimension with the proper number of processors. This modification is similar to using `NicolPlus` to solve the heterogeneous array partitioning problem [36]. Let us call this algorithm `JAG-M-ALLOC`. The overall algorithm has a complexity of $O((P \log \frac{n_1}{P} \max_S Q_S \log \frac{n_2}{Q_S})^2)$.

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 $n_1 m$ calls to L_{max} to evaluate, and at most $n_1^2 m$ calls to $1D$ to evaluate. Evaluating one function call of L_{max} can be done in $O(n_1 m)$ and evaluating $1D$ can be done in $O((x \log \frac{n_2}{x})^2)$ using the algorithm from [31]. 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 in practice. 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 (Initially given by a heuristic such as `JAG-M-HEUR` or found during the exploration of the search space), 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.

2.3.2 Hierarchical Bipartition

Hierarchical bipartitioning techniques consist of obtaining partitions that can be recursively generated by splitting one of the dimensions in two intervals. An example of such a partition is depicted in Figure 2.1(d). Notice that such partitions can be represented by a binary tree for easy indexing. We present first `HIER-RB`, a known algorithm to generate hierarchical bipartitions. Then we propose `HIER-OPT`, an original optimal dynamic programming algorithm. Finally, a heuristic algorithm, called `HIER-RELAXED` is derived from the dynamic programming algorithm.

A classical algorithm to generate hierarchical bipartition is Recursive Bisection which has originally been proposed in [5] 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 $\left\lfloor \frac{m}{2} \right\rfloor$ processors and the other part $\left\lfloor \frac{m}{2} \right\rfloor + 1$ 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 [41]. Another variant decides which direction to cut by selecting the direction with longer length.

We now propose `HIER-OPT`, 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.

$$L_{max}(x_1, x_2, y_1, y_2, m) = \min_j \min \Big\{ \tag{2.3.1}$$

$$\min_x \max\{L_{max}(x_1, x, y_1, y_2, j), L_{max}(x+1, x_2, y_1, y_2, m-j)\}, \tag{2.3.2}$$

$$\min_y \max\{L_{max}(x_1, x_2, y_1, y, j), L_{max}(x_1, x_2, y+1, y_2, m-j)\}\Big\} \tag{2.3.3}$$

Equation 2.3.2 considers the partition in the first dimension and Equation 2.3.3 considers 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 naïvely be evaluated in $O((x_2 - x_1 + y_2 - y1)m)$. Notice that optimization techniques similar to the one used in Section 2.3.1 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 $O(m^2 \log(\max(n_1, n_2))))$.

2.3.3 More General Partitioning Schemes

The considerations on Hierarchical Bipartition can be extended to any kind of recursively defined pattern such as the ones presented in Figures 2.1(e) and 2.1(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 a dynamic programming technique. The number of functions to evaluate will keep being in $O(n_1^2 n_2^2 m)$; one function for each sub rectangle and number of processors..

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 serve as a basis to derive 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 [32] provides a 2-approximation algorithm which heavily relies on linear programming.

2.4 Experimental Evaluation

2.4.1 Experimental Setting

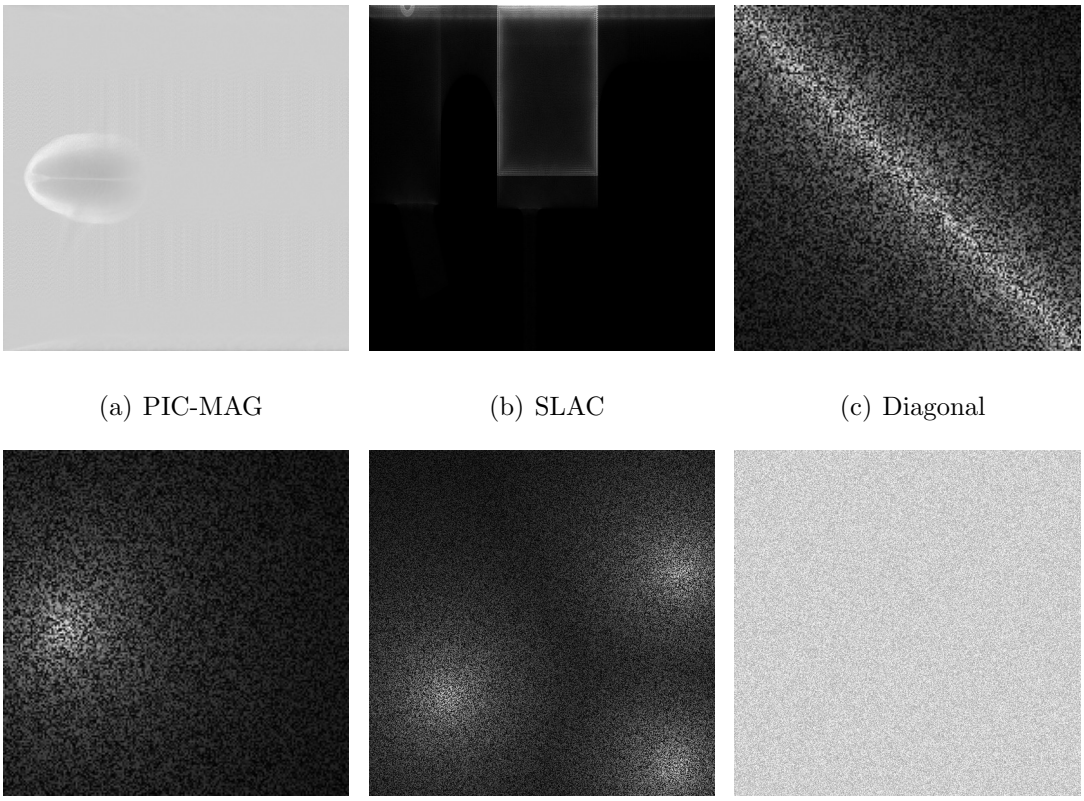
This section presents an experimental study of the main algorithms. For rectilinear partitions, both the uniform partitioning algorithm `RECT-UNIFORM` and `RECT-NICOL` algorithm have been implemented. For $P \times Q$-way and m-way jagged partitions, the following heuristics and optimal algorithms have been implemented: `JAG-PQ-HEUR`, `JAG-PQ-OPT-NICOL`, `JAG-PQ-OPT-DP`, `JAG-M-HEUR`, `JAG-M-HEUR-PROBE` and `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++ (see Chapter 4 for details). 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 [19]. 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 2.2(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 2.2(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 [31].



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

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

Figure 2.2: Examples of real and synthetic instances.

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 2.2(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 2.2(c)), the reference point is the closest point on the diagonal of the matrix. In peak (Figure 2.2(d)), the reference point is one point chosen randomly at the beginning of the execution. In multi-peak (Figure 2.2(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 [28].

The performance of the algorithms is given using the load imbalance metric defined in Section 2.1. For synthetic dataset, the load imbalance is computed over 10 instances as follows: $\frac{\sum_I L_{max}(I)}{\sum_I L_{avg}(I)} - 1$. 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 algorithms.

2.4.2 Jagged algorithms

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 load imbalance of the three variants are quite close and the orientation

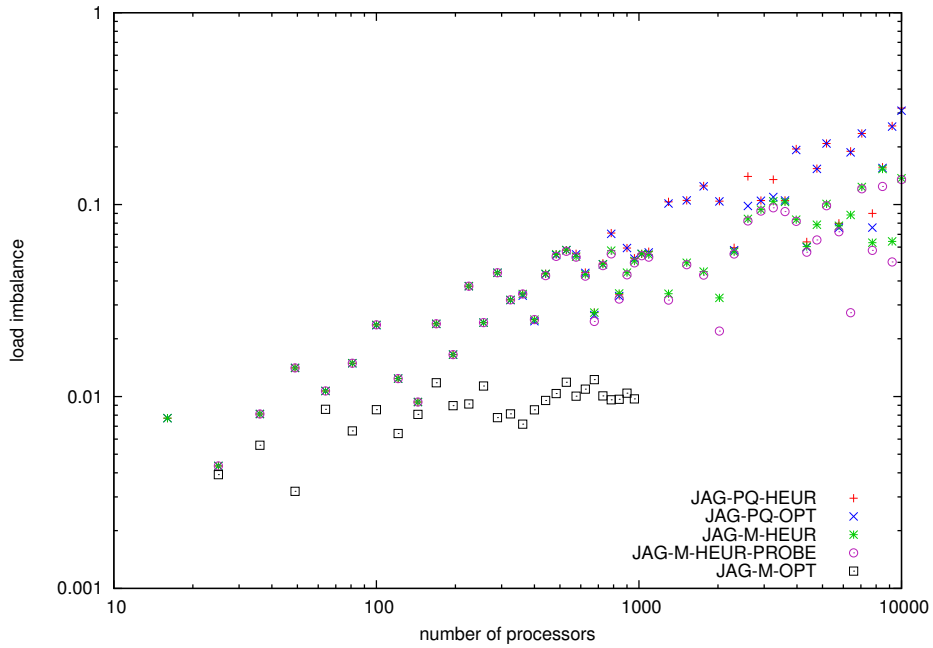


Figure 2.3: Jagged methods on PIC-MAG iter=30,000.

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 the m-way jagged partitioning heuristics are as fast as heuristic jagged partitioning, trying both dimensions and taking the one with best load imbalance is a good option. From now on, if the variant of a jagged partitioning algorithm is unspecified, we will refer to their `BEST` variant.

We proposed in Section 2.3.1 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 2.3 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.

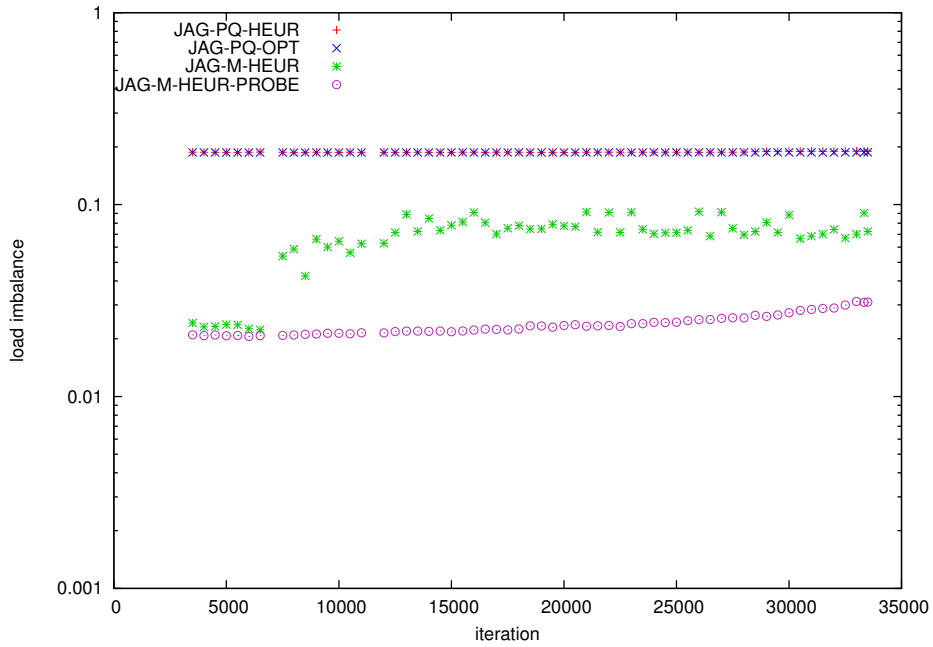


Figure 2.4: Jagged methods on PIC-MAG with $m = 6400$.

On less than one thousand processors, `JAG-M-HEUR`, `JAG-PQ-HEUR`, `JAG-PQ-OPT` and `JAG-M-HEUR-PROBE` 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 $P \times Q$-way jagged heuristic. `JAG-M-HEUR-PROBE` usually obtains the same load imbalance that `JAG-M-HEUR` or does slightly better. But on some cases, it leads to dramatic improvement. One can remark that the m-way jagged heuristics always reaches a better load balance than the $P \times Q$-way jagged partitions.

Figure 2.4 presents the load imbalance of the algorithms with 6,400 processors for the different iterations of the PIC-MAG application. $P \times Q$-way jagged partitions have a load imbalance of 18% while the imbalance of the partitions generated by

JAG-M-HEUR varies between 2.5% (at iteration 5,000) and 16% (at iteration 18,000). JAG-M-HEUR-PROBE achieves the best load imbalance of the heuristics between 2% and 3% on all the instances.

In Figure 2.3, 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 the PIC-MAG application on 1,000 processors. This value is much smaller than the 6% imbalance of JAG-M-HEUR and JAG-M-HEUR-PROBE. 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. Notice that an optimal m-way partition with a given number of columns could be computed optimally using dynamic programming. Figure 2.5 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, JAG-M-HEUR and JAG-M-HEUR-PROBE make some waves on Figure 2.3 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 2.5.

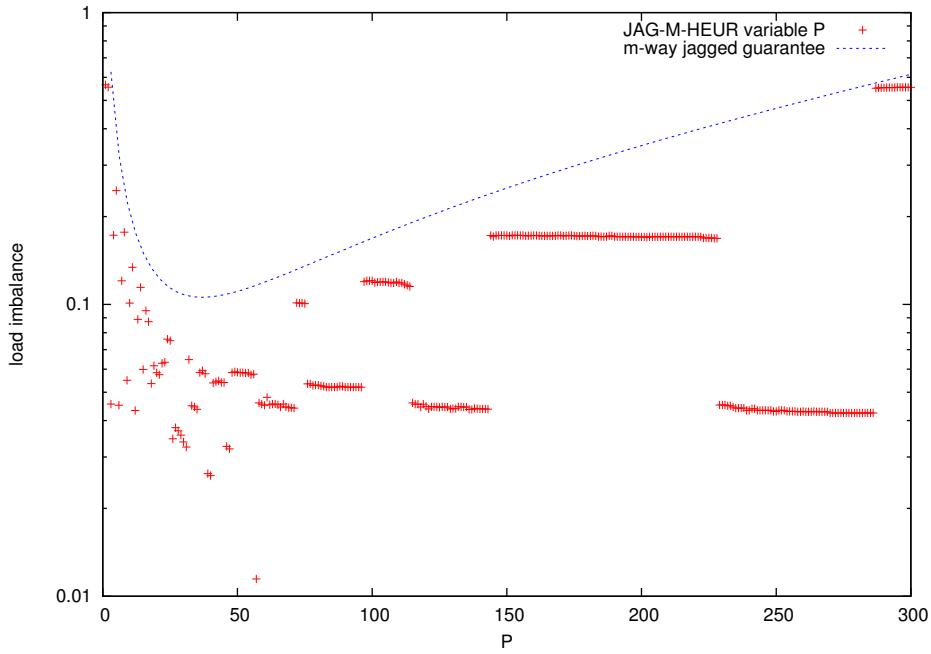


Figure 2.5: Impact of the number of stripes in `JAG-M-HEUR` on a 514x514 Uniform instance with $\Delta = 1.2$ and $m = 800$.

2.4.3 Hierarchical Bipartition

There are four variants of `HIER-RB` depending on the dimension that will be partitioned in two. In general the load imbalance increases with the number of processors. The `HIER-RB-LOAD` variant achieves a slightly smaller load balance than the `HIER-RB-HOR`, `HIER-RB-VER` and `HIER-RB-DIST` variants. The results are similar on all the classes of instances and are omitted.

There are also four variants to the `HIER-RELAXED` algorithm. Figure 2.6 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 `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

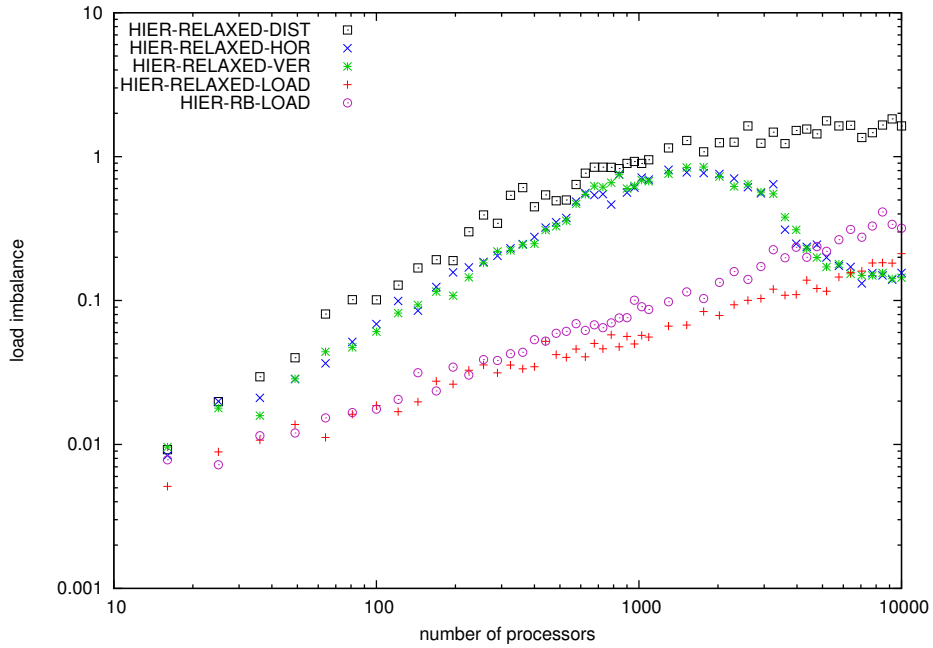


Figure 2.6: `HIER-RELAXED` on 512x512 Multi-peak.

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 2.7 shows this behavior.

Since the load variant of both algorithm leads to the best load imbalance, we will refer to them as `HIER-RB` and `HIER-RELAXED`.

We proposed in Section 2.3.2, `HIER-OPT`, a dynamic programming algorithm to compute the optimal hierarchical bipartition. We did not implement `HIER-OPT` since we expect it to run in hours even on small instances. However, we derived `HIER-RELAXED`, from the dynamic programming formulation. Figure 2.6 and 2.7 include the performance of `HIER-RB` and allow to compare it to `HIER-RELAXED`. It is

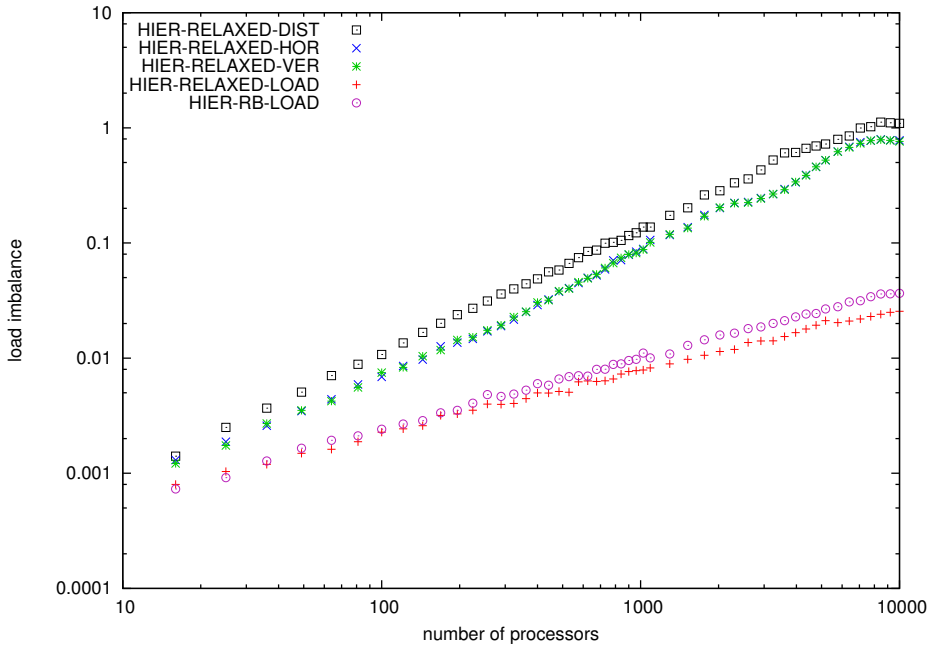


Figure 2.7: `HIER-RELAXED` on 4096x4096 Diagonal.

clear that `HIER-RELAXED` leads to a better load balance than `HIER-RB` in these two cases. However, the performance of `HIER-RELAXED` might be very erratic when the instance changes slightly. For instance, on Figure 2.8 the performance of `HIER-RELAXED` during the execution of the PIC-MAG application is highly unstable.

2.4.4 Execution time

In all optimization problems, the trade-off between the quality of a solution and the time spent computing it appears. We present in Figure 2.9 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 second even on 10,000 processors. The

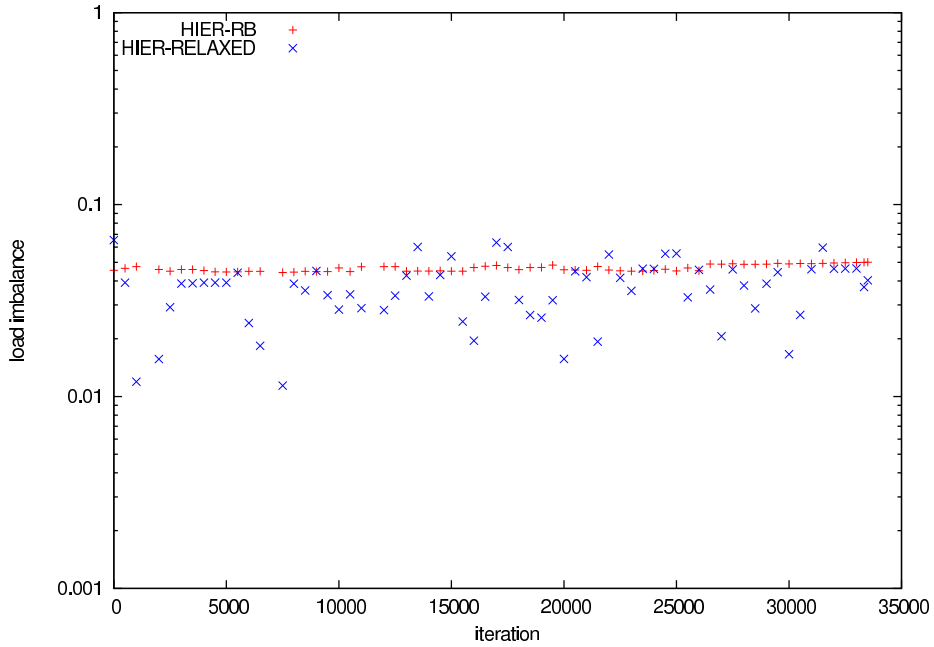


Figure 2.8: Hierarchical methods on PIC-MAG with $m = 400$.

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. Notice that the execution of `JAG-M-HEUR-PROBE` takes about twice longer than `JAG-M-HEUR`. 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.

Two algorithms are available to compute the optimal $P \times Q$-way jagged partition. Despite the various optimizations implemented in the dynamic programming algorithm, `JAG-PQ-OPT-DP` is about one order of magnitude slower than `JAG-PQ-OPT-NICOL`.

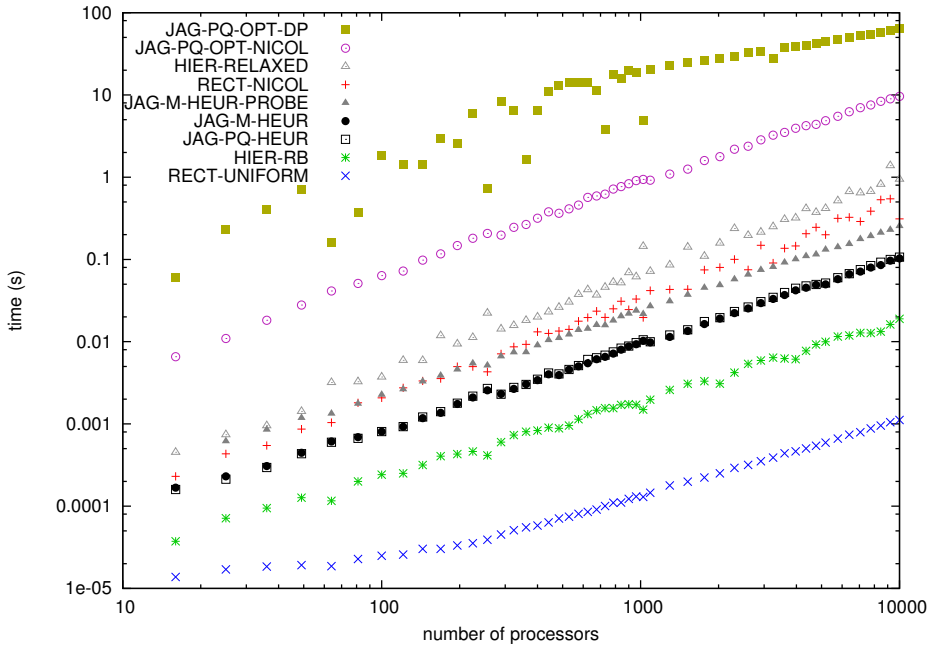


Figure 2.9: Runtime on 512x512 Uniform with $\Delta = 1.2$.

`JAG-PQ-OPT-DP` takes 63 seconds to compute the solution on 10,000 processors whereas `JAG-PQ-OPT-NICOL` only needs 9.6 seconds. Notice that using heuristic algorithm `JAG-PQ-HEUR` is two order of magnitude faster than `JAG-PQ-OPT-NICOL`, the fastest known optimal $P \times Q$-way jagged algorithm.

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 31 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 algorithms 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 is required by all two

dimensional algorithms. Hence, the time taken by these operations is not included in the presented timing results. For reference, it is about 40 milliseconds on a 512x512 matrix.

2.4.5 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 partitioning techniques provide better solutions 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-PROBE`, `HIER-RB` and `HIER-RELAXED`. As a point of reference, the results presented in this section also include the result of algorithm generating rectilinear partitioning, namely, `RECT-UNIFORM` and `RECT-NICOL`.

Figure 2.10 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%. `RECT-NICOL` reaches 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-PROBE` outperforms all the other algorithms by providing a constant 5% load imbalance.

Figure 2.11 shows the performance of the algorithms while varying the number of processors at iteration 20,000. The conclusions on `RECT-UNIFORM`, `RECT-NICOL` and `HIER-RB` stand. Depending on the number of processors, the performance of

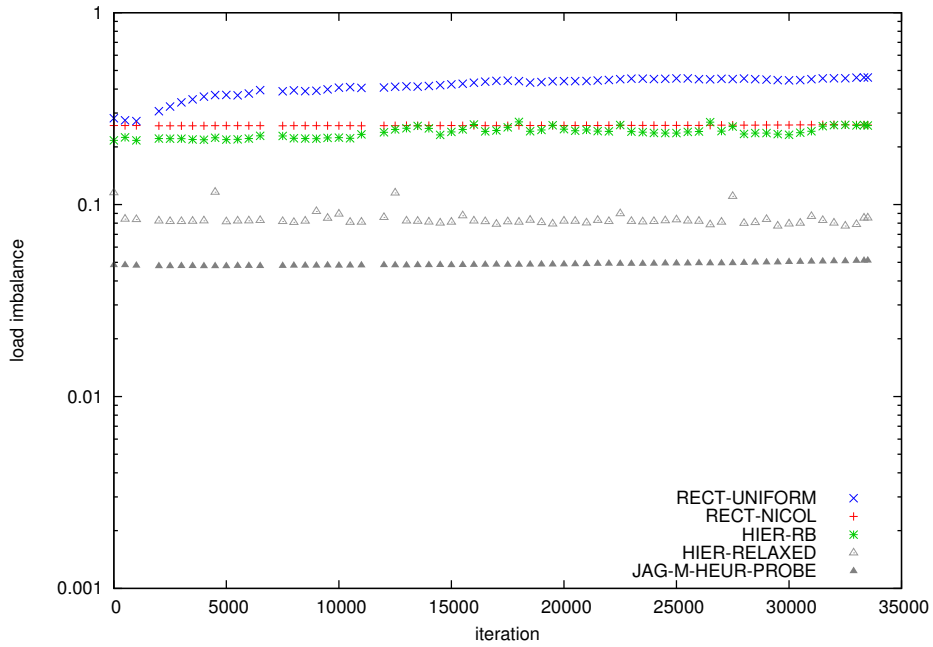


Figure 2.10: Main heuristics on PIC-MAG with $m = 9216$.

`JAG-M-HEUR-PROBE` varies and in general `HIER-RELAXED` leads to the best performance, in this test.

Figure 2.12 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-PROBE`, 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 2.8) and because of its not-that-low running time (up to one second on 10,000 processors according to Figure 2.9). `JAG-M-HEUR-PROBE` seems much a more stable heuristic.

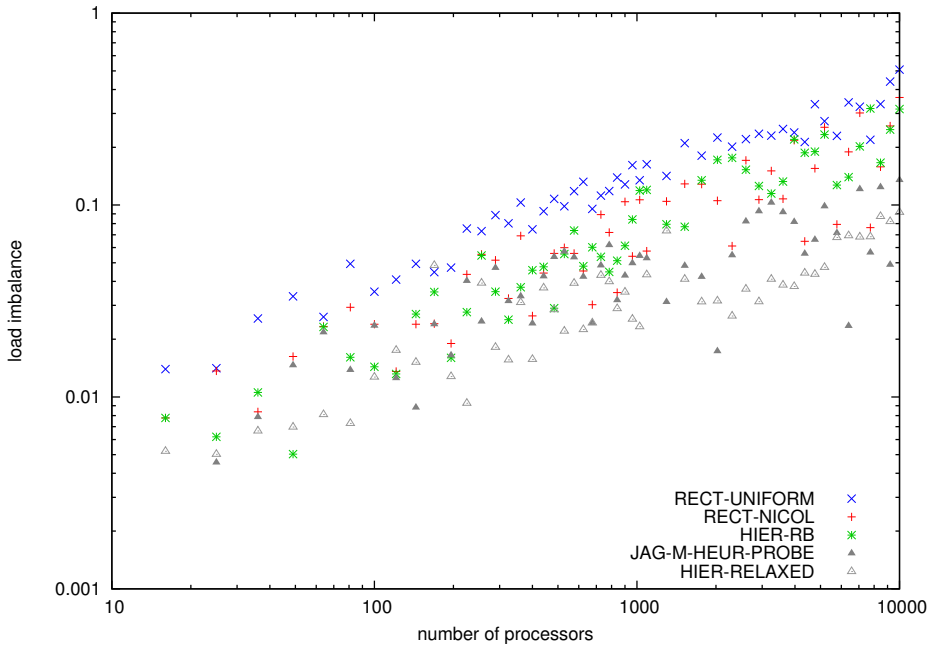


Figure 2.11: Main heuristics on PIC-MAG iter=20,000.

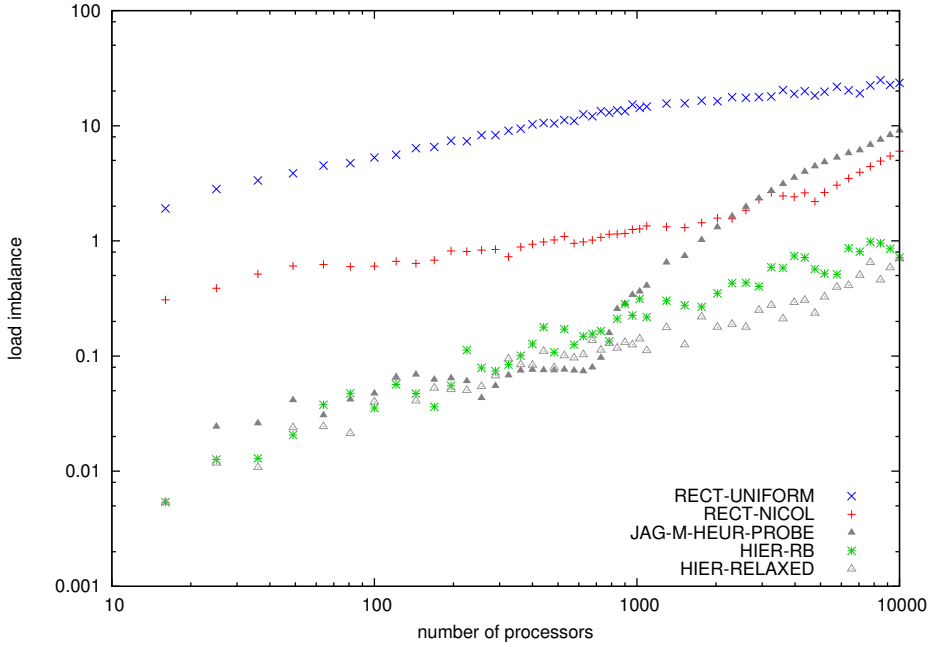


Figure 2.12: Main heuristics on SLAC.

The bad load balance it presents on Figure 2.11 is due to a badly chosen number of partitions in the first dimension.

2.5 Hybrid partitioning scheme

The previous sections show that we have on one hand, heuristics that are good and fast, and on the other hand, optimal algorithms which are even better but to slow to be used in most practical cases. This section presents some engineering techniques one can use to obtain better results than using only the heuristics while keeping the runtime of the algorithms reasonable.

Provided, in general the maximum load of a partition is given by the most loaded rectangle and not by the general structure of the partition, one idea is to use the optimal algorithm to be locally efficient and leave the general structure to a faster algorithm. We introduce the class of `HYBRID` algorithms which construct a solution in two phases. A first algorithm will be used to partition the matrix A in P parts. Then the parts will be independently partitioned with a second algorithm to obtain a solution in m parts. This section investigates the hybrid algorithms and try to answer the following questions: Which algorithms should be used at phase 1 and phase 2? In how many parts the matrix should be divided in the first phase (i.e., what should P be)? How to allocate the m processors between the P parts? And most importantly, is there any advantage in using hybrid algorithms?

Between the two phases, it is difficult to know how to allocate the m processors to the P parts without doing a deep search. We choose to allocate the parts proportionally according to the rule used in `JAG-M-HEUR`, i.e., each rectangle r will first be allocated $Q_r = \lceil \frac{L(r)}{L(A)}(m-P) \rceil$ parts. The remaining processors are distributed greedily.

We conducted experiments using different PIC-MAG instances. All the values of

P were tried between 2 and $\frac{m}{2}$. We will denote the `HYBRID` algorithm using `ALGO1` for phase 1 and `ALGO2` for phase 2 as `HYBRID(ALGO1/ALGO2)`.

The first round of experiments mainly showed three observations. (No results are shown since similar results will be presented later.) First, `HYBRID` is too slow to use `JAG-M-OPT` at the second phase for studying the performance (e.g., partitioning PIC-MAG at iteration 5000 on 1024 processors using $P = 17$ takes 78 seconds). Second, the performance shows "waves" when P varies which are correlated with the values of $\lceil \frac{m-P}{P} \rceil$. Finally `HYBRID` can obtain load imbalances better than `JAG-M-HEUR` and `HIER-RELAXED` on some configuration confirming that `HYBRID` might be useful.

To make the algorithm faster, we introduce the notion of *fast* and *slow* algorithms at phase 2. The *fast* algorithm is first run on each part and the parts are sorted according to their maximum load. The *slow* algorithm is run on the part of higher maximum load. If the solution returned by the *slow* algorithm improves the maximum load of that part, the solution is kept and the parts are sorted again. Otherwise, the algorithm terminates. This modification increased the speed of the algorithm up to an order of magnitude. (Using `JAG-M-HEUR-PROBE` as the *fast* algorithm in phase 2 allows to run PIC-MAG at iteration 5000 on 1024 processors using $P = 17$ in 38 seconds, halving the computation time.) Detailed timing on PIC-MAG at iteration 5000 using 1024 processors can be found in Figure 2.13. The `HYBRID(JAG-M-HEUR/JAG-M-OPT)` curve is the original implementation of `HYBRID` using `JAG-M-HEUR` at phase 1 and `JAG-M-OPT` at phase 2. The `HYBRID-F(JAG-M-HEUR/JAG-M-OPT)` curve presents the timing obtained with the use of *fast* and *slow* algorithm. The `HYBRID` algorithm using `JAG-M-OPT` at both phase is given as a point of reference. All the implementation used `JAG-M-HEUR-PROBE` as the *fast* algorithm. Figure 2.13 shows that using *fast* and *slow* algorithms makes the computation about one order of magnitude faster. Using

JAG-M-OPT at phase 1 is typically orders of magnitude slower than using another algorithm.

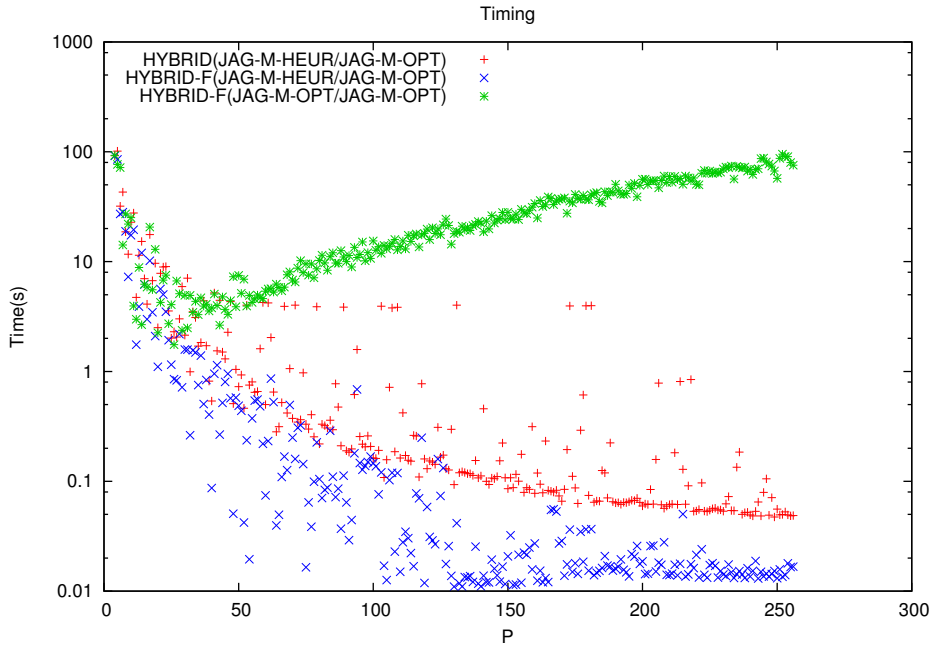


Figure 2.13: Runtime of HYBRID methods on PIC-MAG iter=5000 with $m = 512$.

This improvement allows to run more complete and detailed experiments. In particular, using JAG-M-OPT at phase 2 runs quickly. This allow us to study the performance of HYBRID using an algorithm that get good load balance at phase 2. The load imbalance obtained on PIC-MAG 5000 on 512 are shown in Figure 2.14. Two HYBRID variants that use JAG-M-OPT or HIER-RELAXED at phase 1 are presented. For reference, three horizontal lines present the performance obtained by JAG-M-HEUR, HIER-RELAXED and JAG-M-OPT on that instance. A first remark is that a large number of configurations lead to load imbalances better than JAG-M-HEUR. A significant

number of them get load imbalances better than HIER-RELAXED and sometimes comparable to the performance of JAG-M-OPT. Then, the load imbalance significantly varies with P: it is decreasing by interval which happen to be synchronized with the values of $\lceil \frac{m-P}{P} \rceil$. Finally, the load imbalance is better when the values of P are low. This behavior was predictable since the lower P is the more global the optimization is. However, one should notice that some low load imbalances are found with high values of P.

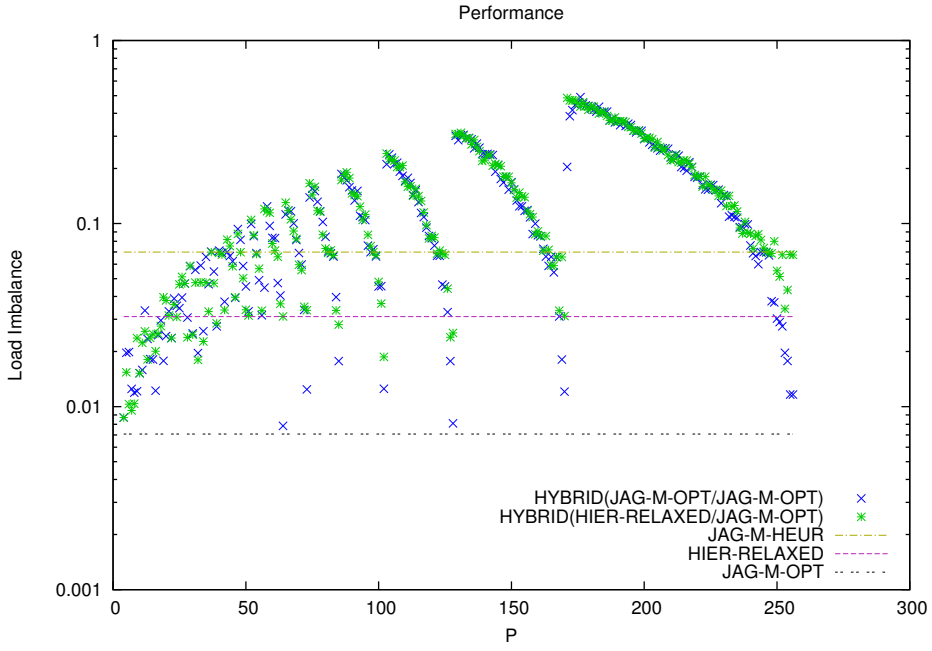


Figure 2.14: Using JAG-M-OPT at phase 2 on PIC-MAG iter=5000 with $m = 512$

Good load imbalance could obviously be obtained by trying every single value of P. However, such a procedure is likely to take a lot of time. Provided the phase 2 algorithm takes a large part of the computation time, it will be interesting to predict the performance of the second phase without having to run it. We define the

expected load imbalance as the load imbalance that would be obtained provided the second phase balances the load optimally, i.e., $eLI = \max_r \frac{L(r)}{Q_r}$. Figure 2.15 presents for each solution its expected load imbalance and the load imbalance obtained once the second phase is run for different values of P. The solutions are presented for two hybrid variants, one using `JAG-M-HEUR` at phase 2 and the other one using `JAG-M-OPT`. For similar expected load imbalance, the load imbalance obtained using `JAG-M-HEUR` are spread over an order of magnitude. However, the load imbalance obtained by `JAG-M-OPT` are much more focused. The expected load imbalance and obtained load imbalance are well correlated when `JAG-M-OPT` is used at phase 2.

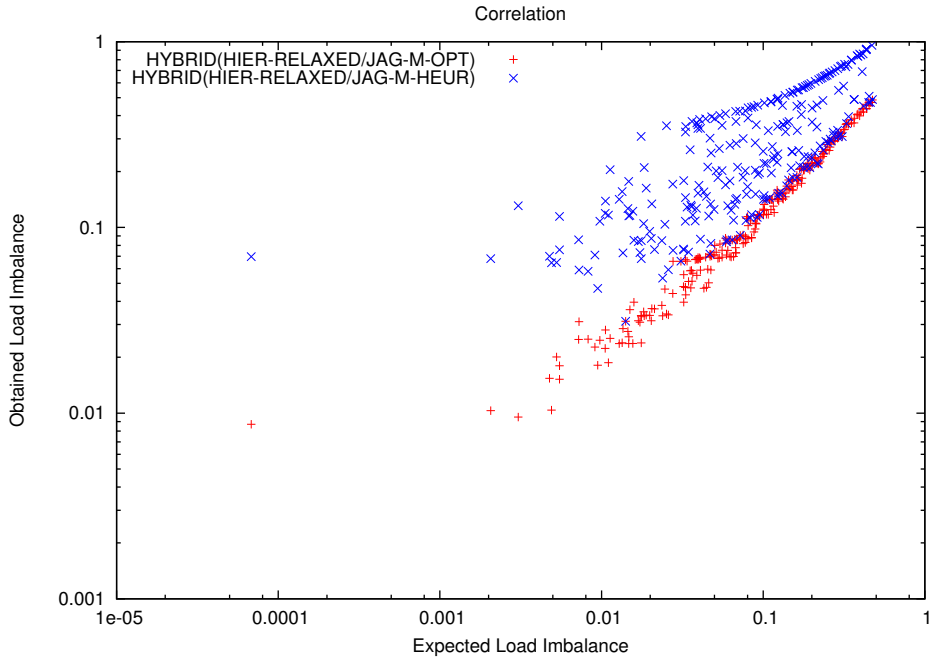


Figure 2.15: Correlation between expected load imbalance at the end of phase 1 and the obtained load imbalance on PIC-MAG iter=5000 with $m = 512$

The previous experiments show two things. The actual performance are correlated with expected performance at the end of phase 1 if `JAG-M-OPT` is used in phase 2. The

load imbalance decreases in an interval of values of P synchronized with the values of $\lceil \frac{m-P}{P} \rceil$. Therefore, we propose to enumerate the values of P at the end of such intervals. For each of these value, the phase 1 algorithm is used and the expected load imbalance is computed. The phase 2 is only applied on the value of P leading to the best expected load imbalance. Obviously the best expected load imbalance will be given by the non-hybrid case $P = 1$, but it will lead to a high runtime. The tradeoff between the runtime of the algorithm and the quality of the solution should be left to the user by specifying a minimal P.

Figure 2.16 presents the load imbalance obtained on the PIC-MAG datasets at iteration 10000 using $\sqrt{m}$ as minimal P. The `HYBRID` algorithm obtains a load balance usually better than `JAG-M-HEUR-PROBE` and often better than `HIER-RELAXED`. The algorithm leading to the best load imbalance seems to depend on the number of processors. For instance, Figure 2.17 shows the load imbalance of the algorithms on 7744 processors. `JAG-M-HEUR-PROBE` leads constantly to better results than `HIER-RELAXED`. And `HYBRID` typically improve both by a few percents.

However, on 6400 processors (Figure 2.18), `HYBRID` almost constantly improves the result of `HIER-RELAXED` by a few percents but does not achieve better load imbalance than `JAG-M-HEUR-PROBE`. Figure 2.4 showed that `JAG-M-HEUR` is significantly outperformed by `JAG-M-HEUR-PROBE` in that configuration. Recall that the main difference between these heuristics is that the former distributes the processors among the stripes only based on the load of each stripe while the latter use the minimum number of processors per stripe to obtain the minimum load balance. The same idea could be applied to `HYBRID` algorithms looking for the minimum number of processors to allocate to each part without degrading the load imbalance and use these processors on the parts that lead to the maximum load. This modification will improve the load imbalance but will also increase the running time significantly.

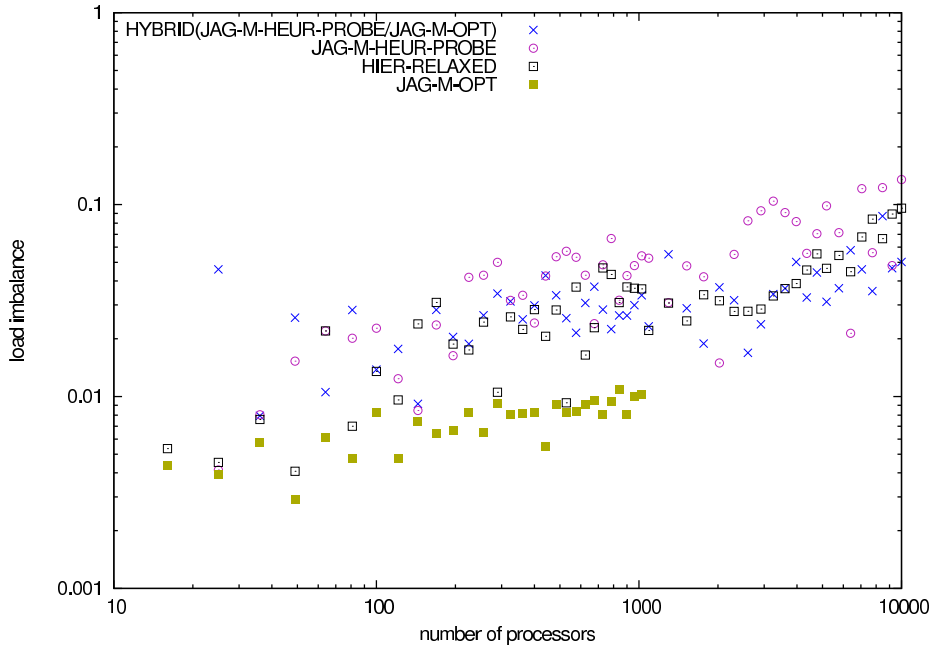


Figure 2.16: `HYBRID` algorithm on PIC-MAG iter=10000

The runtime of the algorithm is presented in Figure 2.19. It shows that the `HYBRID` algorithm is two or three orders of magnitude slower than the heuristics but one to two orders of magnitude faster than `JAG-M-OPT`. However, `HYBRID` algorithms are likely to parallelize pleasantly.

Some more engineering techniques could be applied to `HYBRID`. Different time/quality tradeoff could be obtained by stopping the use of the *slow* algorithm in phase 2 when the improvement become smaller than a given threshold. Using a 3-phase `HYBRID` mechanism could be another way of obtaining different trade-offs.

`HYBRID` is not the only option for algorithm engineering. One idea that might lead to interesting time/quality tradeoff would be to avoid running dynamic programming algorithms all the way through. Early termination can be decided based on a time allocation or a targeted maximum load.

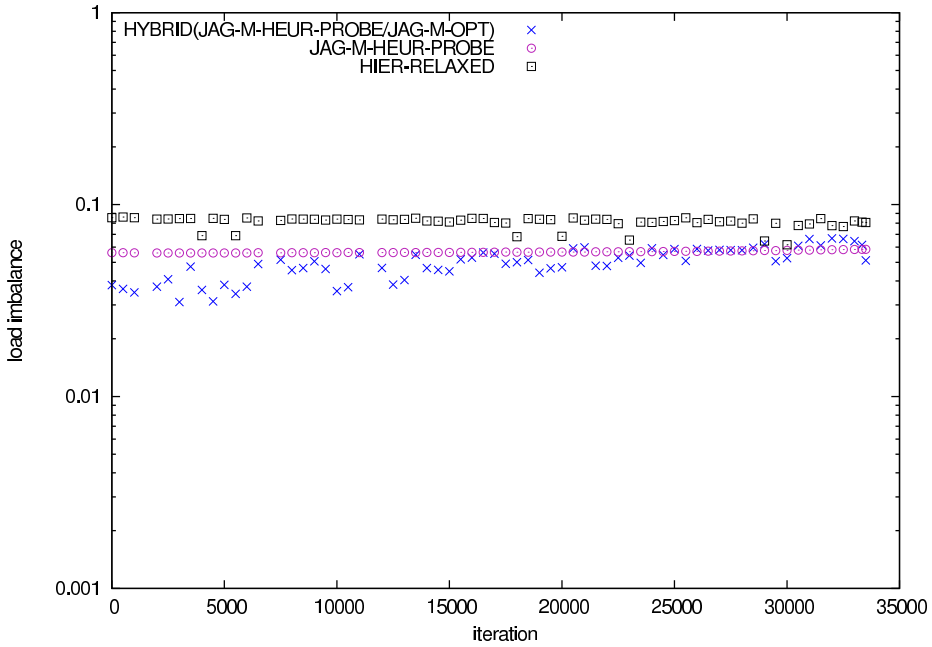


Figure 2.17: `HYBRID` algorithm on PIC-MAG on 7744 processors

Finally, different kind of iterative improvement algorithms could be designed. For instance, on m-way jagged partition, `JAG-M-PROBE` provides the optimal number of processors to use in each stripe provided the partition in the main dimension, and `JAG-M-ALLOC` provides the optimal partition in the main dimension provided the number of processors allocated to each stripe. Applying `JAG-M-PROBE` and `JAG-M-ALLOC` the one after the other as long as the solution improves would be one interesting iterative algorithm.

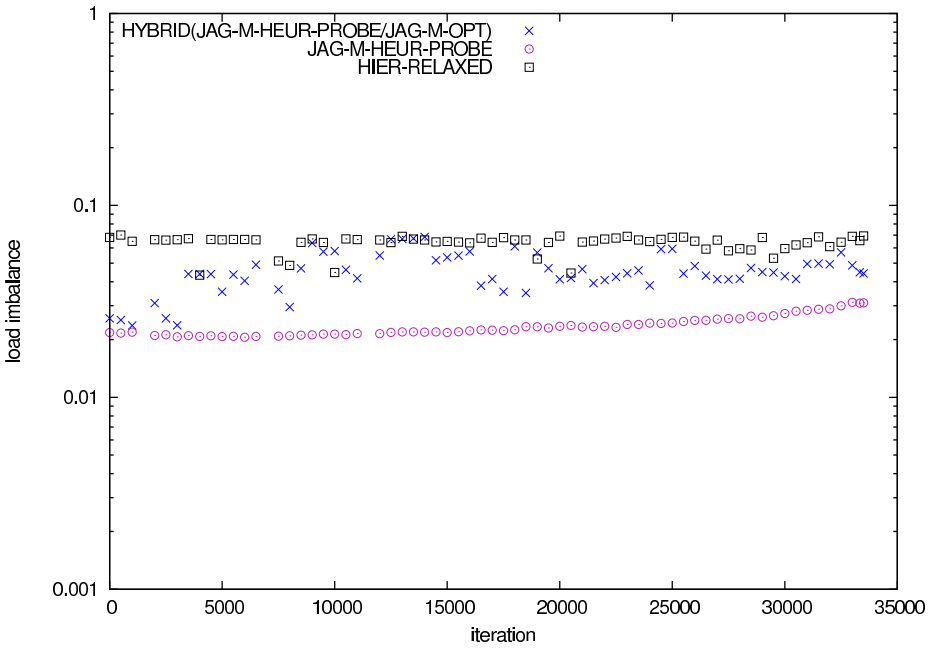


Figure 2.18: `HYBRID` algorithm on PIC-MAG on 6400 processors

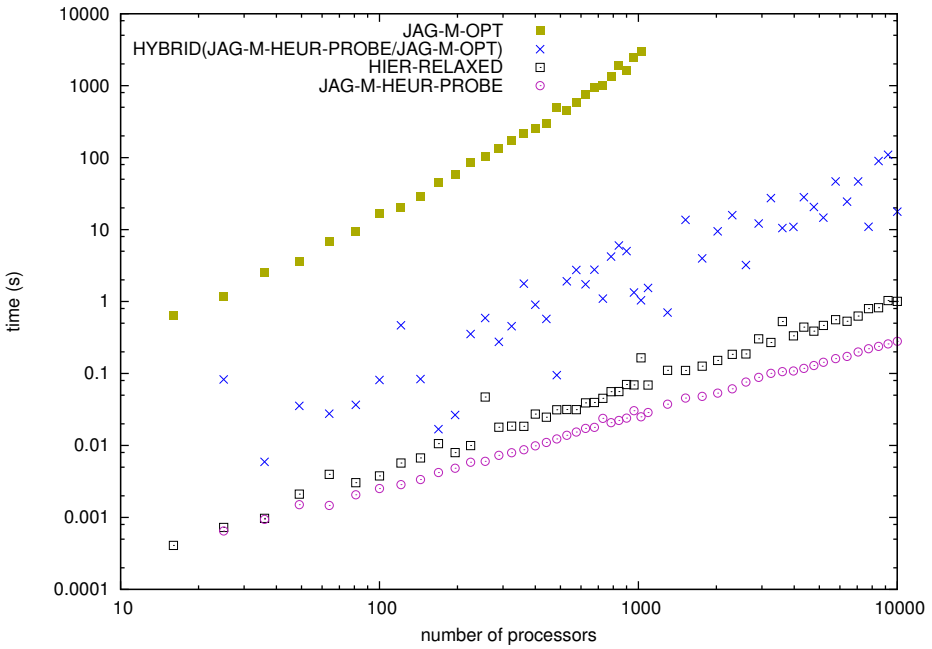


Figure 2.19: Runtime of `HYBRID` methods on PIC-MAG iter=10000

CHAPTER 3

INTER-PROCESSOR COMMUNICATION AND REBALANCING

3.1 Communication Cost

The two dimensional partitioning algorithms that we consider are optimized to minimize the total load of the most loaded processor. However, they may not minimize communication cost. If network latency is much higher than computation time, overall processing will be quite slower until next repartitioning iteration. Both communication and load volume play an important role on overall runtime of stencil operations. Particle-in-cell simulations require tremendous amount of arithmetic stencil operations over matrices. The problem matrix can be partitioned and distributed to reduce total runtime. In this chapter, we will analyze the existing load balancing algorithms in the perspective of communication cost and compare their performance. There are two types of inter-processor communication costs. The first type, called *application cost*, occurs in given partitions. Processors exchange data that is required to obtain the solution of the application. Tasks stay in their assigned processors. In order to minimize application cost, we use *static load balancing* techniques. The second type, called *migration cost*, does not interfere with application itself. The distribution of the load might vary with time and introduce load imbalance. This type of communication cost occurs in re-balancing currently imbalanced partitions.

Re-balancing operation aims to reduce most loaded processor's load by applying a different partition. Tasks are reassigned to different processors if necessary. The technique to minimize this cost is called *dynamic load balancing*.

In any parallel application we would like to minimize overall runtime function. One of the functions given in [9] is as follows.

$$t_{tot} = \alpha(t_{comp} + t_{comm}) + t_{mig} + t_{repart} \tag{3.1.1}$$

where, t_{comp} is computation time, t_{comm} is communication time,

t_{mig} is migration time, and t_{repart} is repartitioning time.

3.2 Problem Definitions

Static load balancing problem is defined as follows: Given an $n_1 \times n_2$ matrix we would like to distribute it to m processors such that total communication cost is minimized. Total communication cost depends on several metrics. The metrics will be discussed in the following sections. The bandwidth assumed to be homogeneous and processors are identical. A processor can only communicate with its neighbors i.e. in a uniform partitioning $P_{i,j}$ can communicate with $P_{i-1,j}, P_{i+1,j}, P_{i,j-1}, P_{i,j+1}$. The cells that lie in the border of two adjacent processors are called connected cells. The cost of communication depends on the size of the message and startup cost of the message. The message transfer occurs in every iteration on borders because of stencil-like operations. Total communication cost $C_{P_i} = \sum_{j=0}^{P} (B_{P_i,P_j}\alpha + \beta)$ where β is the startup cost for a single message, B_{p_i,p_j} is the border length between processor P_i and P_j. P is the neighbor processor count of P_i.

Dynamic load balancing problem is defined as follows: Given an $n_1 \times n_2$ matrix and a partition Pi_1, find a new partition Pi_2 that minimizes both L_{max} and the total load exchange value. Total load exchange of a processor is the sum of load

Figure 3.1: A migration example.

exchange to all other processors in Pi_2. Total data exchange between p_i and P_j is $M_{P_i} = \sum_{j=0}^{P} (L^x_{P_i,P_j}\alpha + \beta)$ where $L^x_{P_i,P_j}$ is the data exchange between processor P_i and P_j. The assumptions on static load balancing applies except that now a processor is allowed to communicate with any other one. For example consider Figure 3.1. In this figure, the color coding indicate partitioning into 4 processors. The top-left processor sends $55+4+10 = 69$ units of data to the top-right processor while the bottom-right processor sends $37 + 6 = 43$ units of data to the bottom-left processor. Then total migration cost of moving from partition $S1$ to $S2$ is $L^x = 69+43 = 112$ units. Notice that in Figure 3.1 $S2$ partition is arbitrarily selected to demonstrate how migration cost is calculated. Neither L_{max} nor L^X is a concern.

We do not consider other topologies such as fat trees, star networks or hypercubes due to complexity of their analysis. In our analysis a processor may have arbitrary number of direct connections. However, a processor can communicate with its immediate neighbors in the 2D partitioning assignment. For instance, all rectilinear partitions correspond to a 2D mesh topology where each processor has 2, 3 or 4 immediate neighbors (assuming $m \geq 4$). In contrast, jagged partitions and hierarchical partitions can have arbitrary number of neighbors.

In our experiments bipartite matching is done greedily. Source and target processors are selected based on the total overlap. Formally, processor P_i sends data to P_j

if the load of $P_i \cap P_j$ is larger than for all $P_i \cap P_x$ for all $x \in S2$ where $S2$ is set of processors in the next iteration's partition.

3.3 Communication Metrics

There are many metrics proposed in the literature [14]. All communication metrics depend on the network topology. We believe that 6 metrics best explain the quality of partitions. *Average neighbor* measures the message volume of the networks with high startup costs. *Maximum neighbor* measures startup cost if network contention is on one component of the network (such as backbone router). *Average border length* measures the data exchange within borders. This is a necessary measure in PIC applications. A cell on the border will exchange its status information with another neighbor processor. Just like maximum neighbor, *Maximum border length* is suitable when we want to measure message volume, and network contention depends on one component of the network. Notice the load of the border line is another metric that we do not calculate in this work.

Generally speaking, if startup costs dominate communication costs, then it makes more sense to use *"maximum"* metrics. Otherwise, if latency is higher, *"average"* metrics should be considered.

Average neighbor

This metric measures the average startup cost of messages that are sent from a processor in an iteration. $V_{avg} = \sum_{i=1}^{m} \frac{V_i}{m}$ where V_p is the neighbor count of processor p.

Maximum neighbor

This metric measures the maximum startup cost in the partition. $V_{max} = \max_{1 \leq p \leq P} V_p$. This metric plays an important role if a component of the network is under too much traffic and becomes a bottleneck.

Average Border Length

This metric measures the average data size in a message. Since connected cells need to exchange messages, the data section of the message will be proportional to the border length of a part. $B_{avg} = \sum_{i=1}^{m} \frac{B_i}{m}$ where B_i is the border length of processor i. The outermost borders of the main matrix are not counted. The borders must reside between two adjacent parts to be counted. Some algorithms such as `JAG-M-HEUR` may not use all m processors. This would affect B_{avg} mentioned earlier. In our experiments, we did not decrease m due to possibly unused processors. The instances that result in some unused processors are very rare and the unused processor number is typically very low.

Maximum Border Length

This metric measures the maximum message length. This metric plays an important role if one component is bottleneck of the network. $B_{max} = \max_{1 \leq p \leq P} B_p$

Average Migration Cost

This metric measures the data exchange made by each processor to move from one partition to another. Notice that globally, total send data is equal to total received data. When calculating this metric we will consider only one of them. $L^x_{avg} = \frac{L^x}{m}$

Maximum Migration Cost

This metric measures the maximum data exchange over all processor to move from one partition to another. Assuming L_p^x is the data sent or received by processor p. Then $L_{max}^x = \max_{1 \leq p \leq P} L_p^x$

3.4 Performance of 2D Algorithms

3.4.1 Uniform Partitioning (`RECT-UNIFORM`)

Uniform partitioning is a very straightforward algorithm. It ensures that the maximum neighbor number is 4 and average neighbor number is very close to but less than 4 (corner processors have 2 neighbors, edge processors have 3). It has very low average border length and maximum border length is optimum. In return to all those good communication values, uniform algorithm leads to high load imbalance as seen in Figure 2.10, 2.11 and 2.12.

3.4.2 Recursive Bisection (`HIER-RB`)

Recursive bisection is a very fast algorithm that works very good on sparse matrix load balancing. On the other hand, the number of neighbors can be quite high as seen in Figure 3.2. It can be reduced by setting up boundaries (with small rectangles). Average and maximum border lengths are higher than rectilinear algorithms and jagged algorithms due to irregularity of partitions in recursive bisection.

3.4.3 Hierarchical Relaxed Bisection (`HIER-RELAXED`)

The relaxed version of hierarchical relaxed bisection leads to distinct one dimensional partitioning. The regularity of one dimensional partitioning results in lower average neighbor values than recursive bisection. Although the load imbalance and average

neighbor are acceptable, hierarchical relaxed bisection is rather bad at all other metrics thus, it should be avoided when communication cost is more important than load imbalance.

3.4.4 Recursive Bisection with Middle Cut (HIER-RB-MIDDLE)

This algorithm is optimized for communication purpose only. It works like recursive bisection with one difference. It geometrically places the cut point in the middle of the sub-rectangles, assigns processors proportional to the load on each side, and finally recurses on each side. This algorithm tries to approximate by changing processor counts rather than cut points.

3.4.5 Nicol's 2D Algorithm (RECT-NICOL)

Nicol's two dimensional algorithm, like uniform partitioning algorithm, limits the number of neighbor processors. Uniform and rectilinear partitioning get exact same values for all communication metrics except for maximum border length. However, Nicol's algorithm beats uniform in load imbalance. Thus Nicol's algorithm should be preferred over uniform. Nicol's algorithm can result in slightly high maximum border length. However, the experiments show that this is a very rare case in uniform matrices.

3.4.6 $P \times Q$-way Jagged (JAG-PQ-HEUR)

According to average and maximum neighbor metric, $P \times Q$-way Jagged algorithm is in a place between m-way jagged and rectilinear partitioning algorithms. This is due to restricted number of processors per row. Therefore $P \times Q$ jagged is slightly better than m-way jagged algorithms. Average and maximum border length is same as the other rectilinear algorithms.

3.4.7 m-way Jagged algorithms JAG-M-HEUR and JAG-M-PROBE

Even though low average and maximum neighbor value, m-way jagged algorithms fail to achieve a low average neighbor value, they are competitive to rectilinear algorithms in all other metrics. In fact, JAG-M-PROBE got the worst results in terms of average neighbor. However, m-way algorithms are still reasonable since maximum neighbor value is lower than hierarchical partitioning algorithms.

3.5 Results

The following 4 charts (Figure 3.2,3.3,3.4,3.5) are performance profiles. It is used when comparing too many test cases with respect to a metric. A performance profile shows the probability that a specific algorithm gives results within some value τ multiple of the best result reached ever all algorithms. Higher τ indicates a better algorithm value. For example HIER-RB-MIDDLE has a point on $\tau = 1.1$ and fraction of 0.7. This means on 70% of the test cases, HIER-RB-MIDDLE did no worse than 1.1 of the best result found by any algorithm in the test. We have tested algorithms on all squared processor numbers between 1 and 10000 (e.g 1, 4, 9, ..., 10000)

The next three charts show degradation of bottleneck. The horizontal axis is snapshots in a time step. The vertical axis shows the load imbalance if we refresh (re-run the algorithm) the partition in every 2, 5 and 10(Figure 3.6, 3.7, 3.8 respectively) iterations. The number of processors is fixed to 1024 which is a reasonable processor count in today's middle sized clusters.

The last four charts concern about the cost of migration. JAG-M-PROBE-HEUR was chosen for migration cost evaluation. Because it is the fastest heuristic algorithm that minimizes the load. Horizontal variant is used in this experiment. As an opponent to this algorithm, RECT-NICOL is selected due to low application communication cost.

The results show that HIER-RB-LOAD is the best algorithm in terms of degradation. In figure 3.6, there are even points that using the same partition makes HIER-RB-LOAD better.

On the other hand JAG-M-HEUR-PROBE-BEST easily jumps to the top values. This is more obvious in longer repartitioning periods shown in Figure 3.6.

JAG-M-HEUR-PROBE-BEST is better than RECT-NICOL in total migration cost (Figures 3.9 and 3.11).

But RECT-NICOL is less chaotic in maximum migration cost. Therefore we can conclude JAG-M-HEUR-PROBE-BEST is acceptable when network component contention is not an issue.

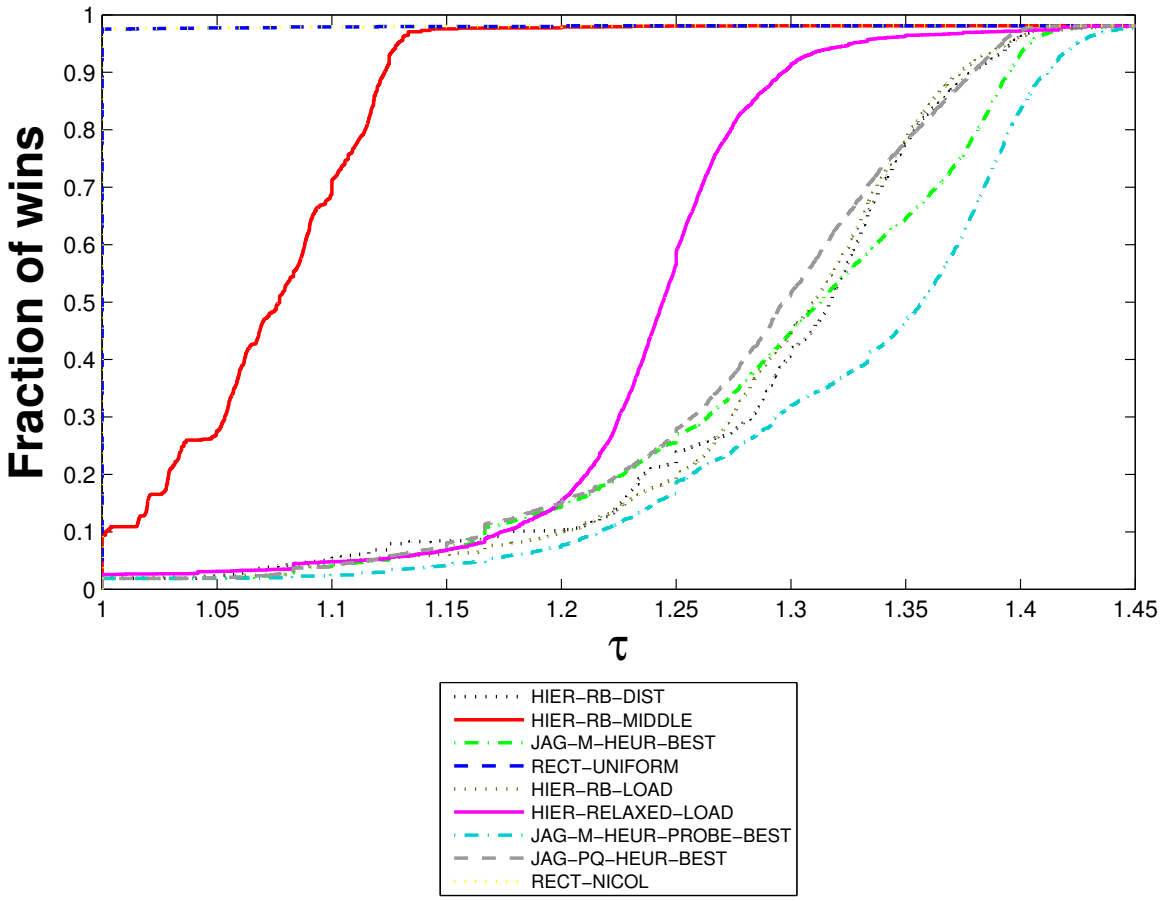


Figure 3.2: Average neighbor performance profile in PICMAC dataset

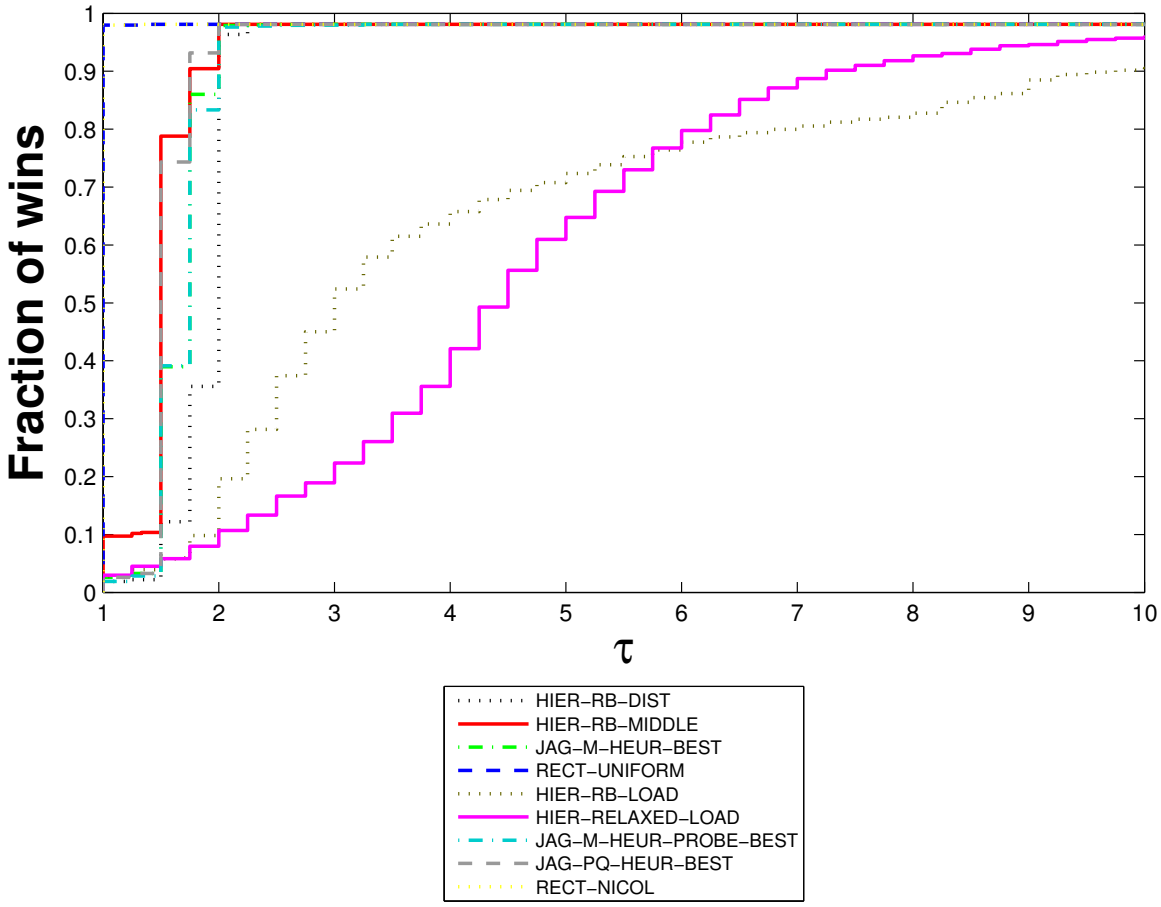


Figure 3.3: Maximum neighbor performance profile in PICMAC dataset

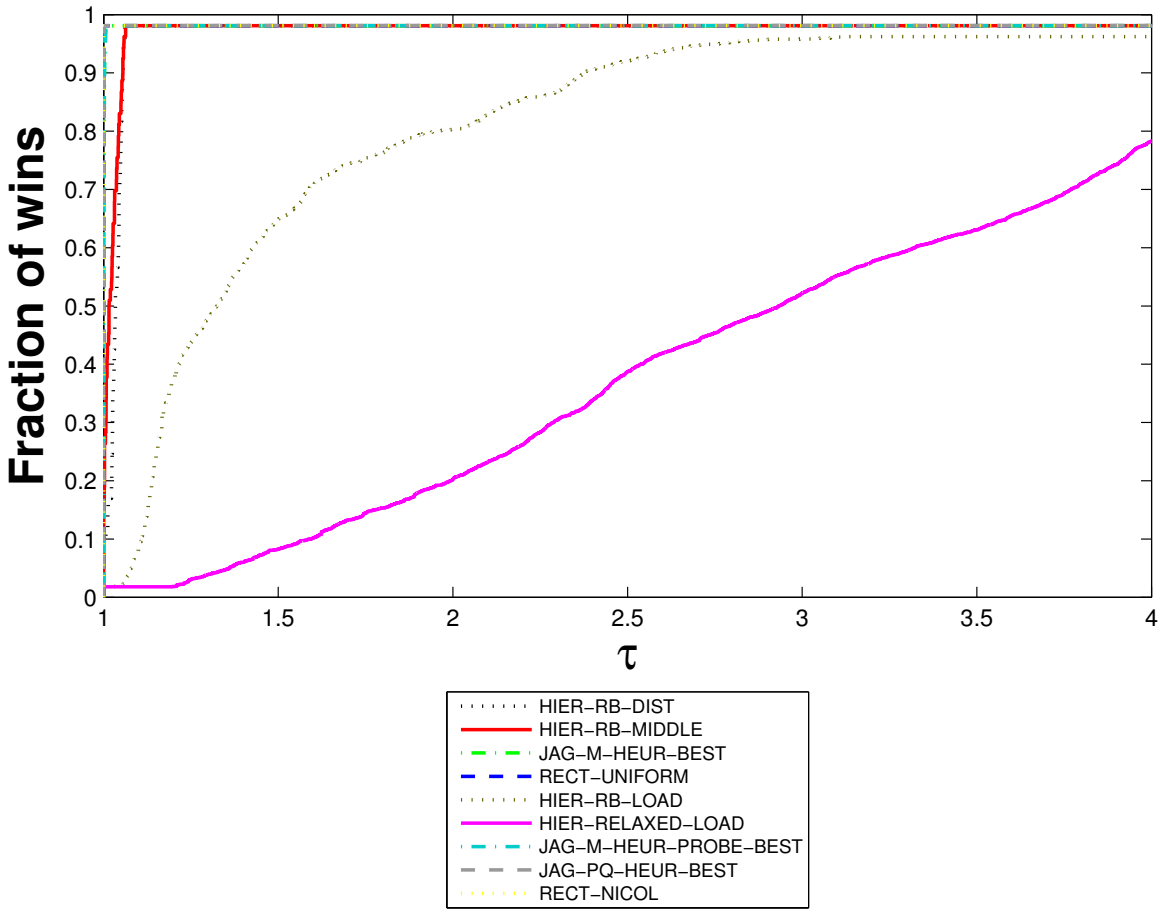


Figure 3.4: Average border length performance profile in PICMAC dataset

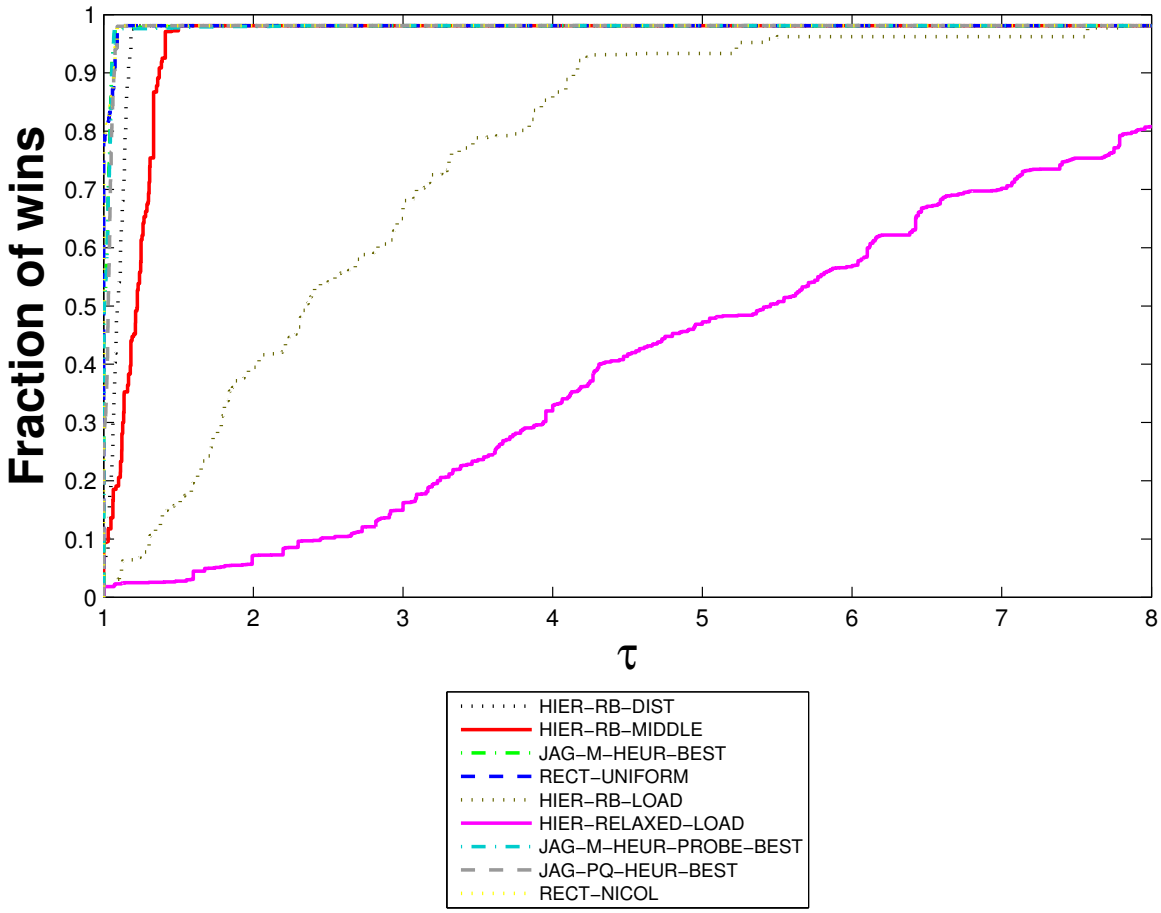


Figure 3.5: Maximum border length performance profile in PICMAC dataset

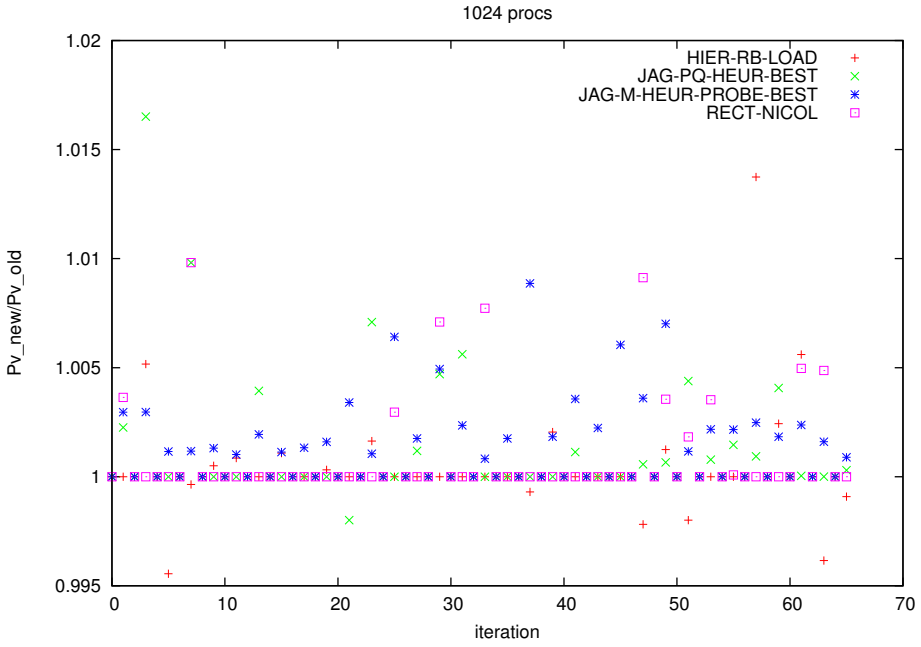


Figure 3.6: Degradation in PICMAC Bottleneck - repartitioned in every 2 iterations

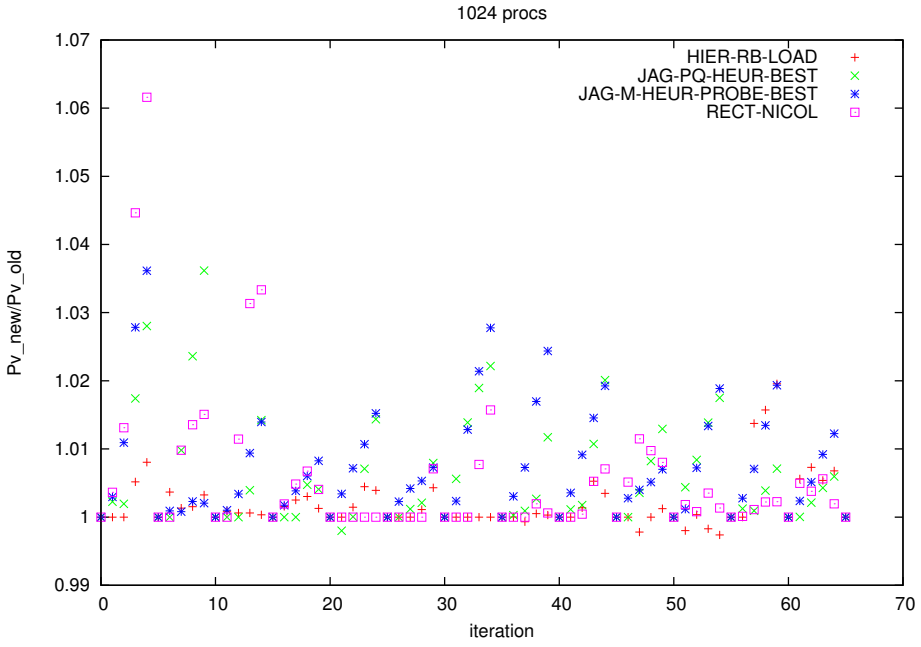


Figure 3.7: Degradation in PICMAC Bottleneck - repartitioned in every 5 iterations

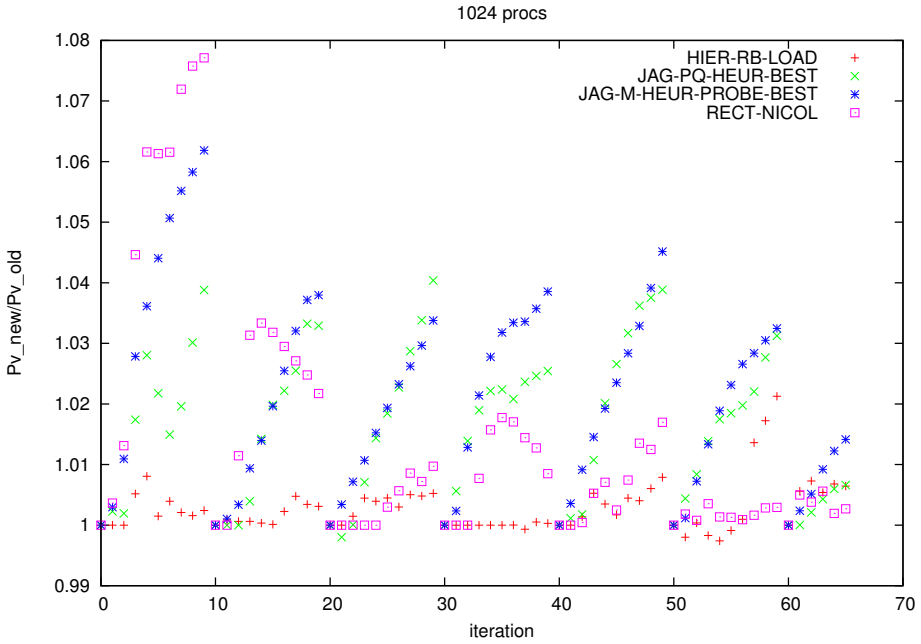


Figure 3.8: Degradation in PICMAC Bottleneck - repartitioned in every 10 iterations

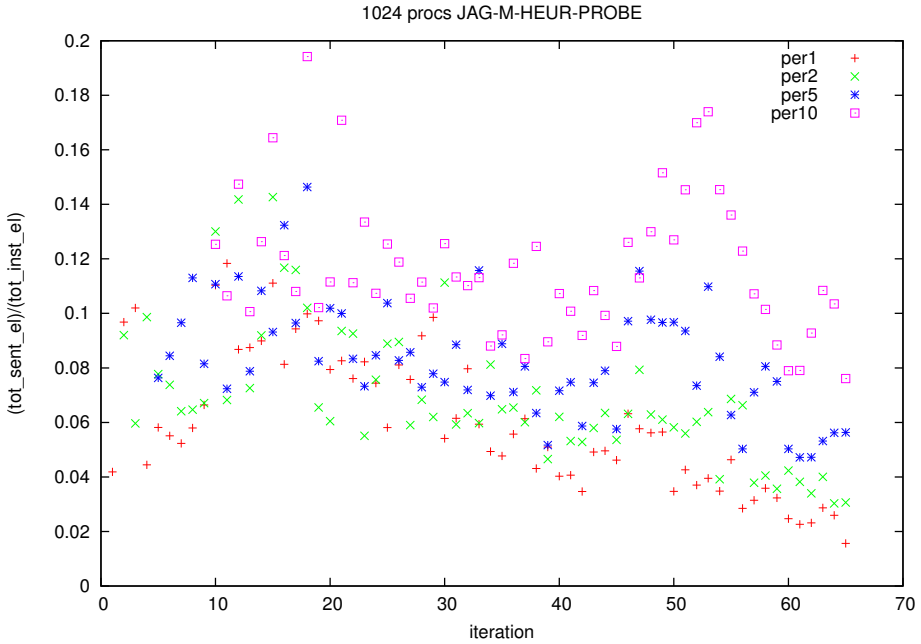


Figure 3.9: JAG-M-HEUR-PROBE total rebalancing cost normalized to instance load

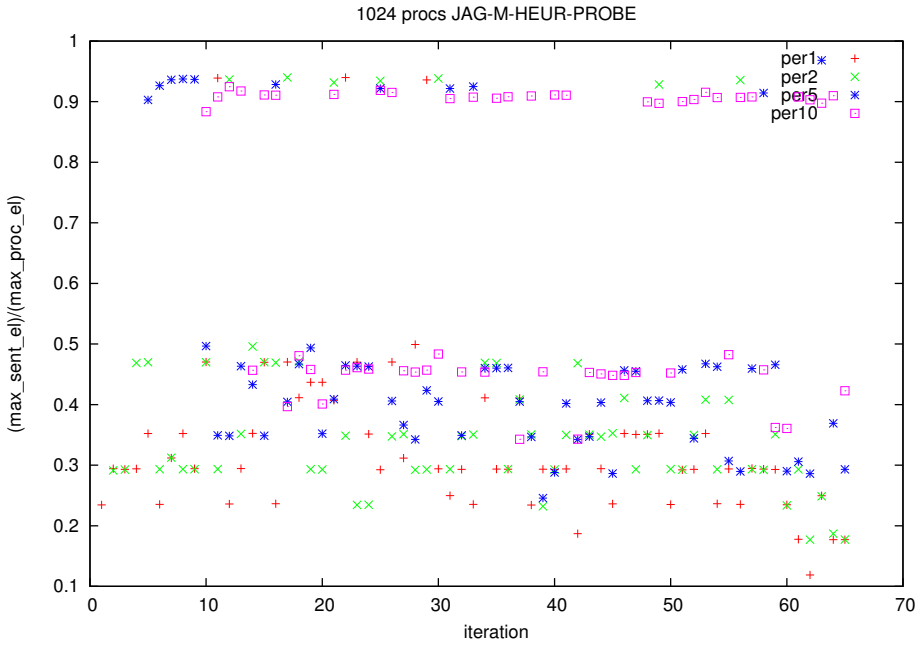


Figure 3.10: JAG-M-HEUR-PROBE maximum send/receive cost normalized to maximum load

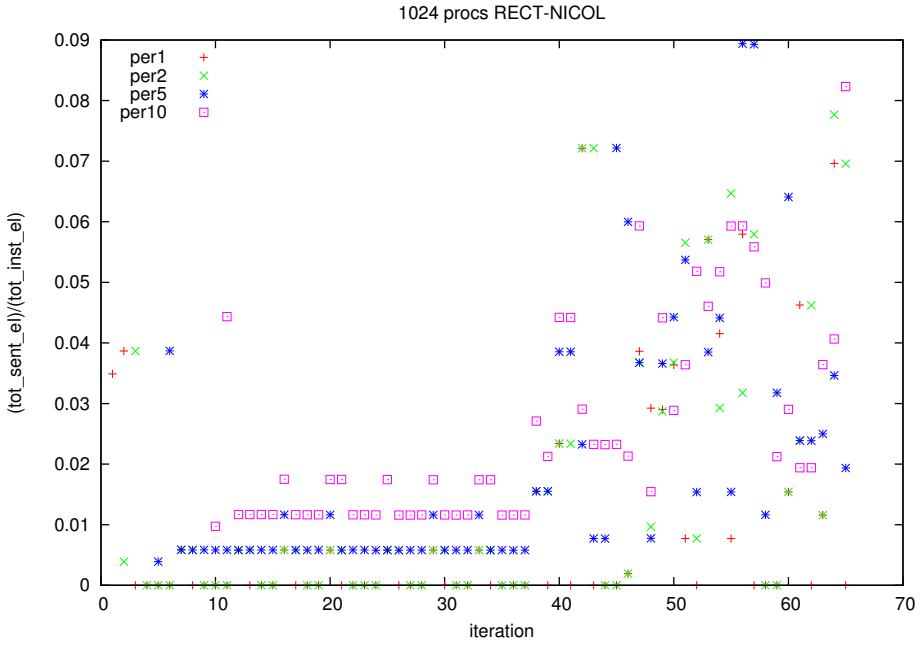


Figure 3.11: RECT-NICOL total rebalancing cost normalized to instance load

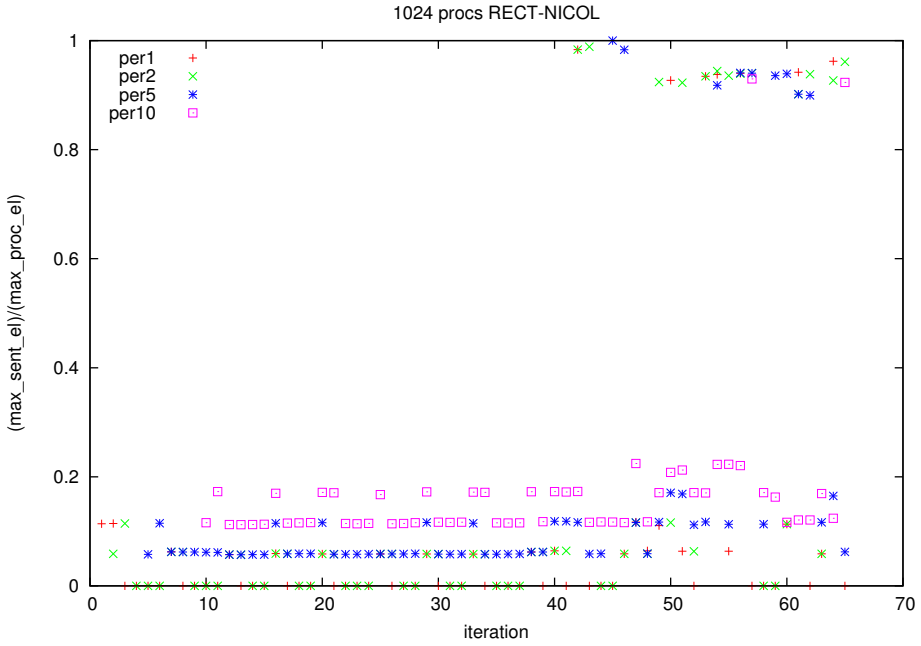


Figure 3.12: RECT-NICOL maximum send/receive cost normalized to maximum load

CHAPTER 4

SOFTWARE

4.1 Overview

The project is written in C++. The project repository has 2 folders. The `data` folder contains example 2D matrix data and source code for matrix synthesizers. The `src` folder contains three important sub-folders: `oned, twod` and `util`. The `oned` folder has all the one dimensional partitioning algorithms. Any 1D algorithm can be easily integrated into 2D algorithms. `twod` contains all the 2D algorithm implementations. `util` contains various auxiliary code for both one and two dimensional partitioning such as rectangle data structure to hold partition information and prefix sum matrix builders. In this chapter, we will explain of the important design features that allow the programmer to easily:

1. Change underlying 1D algorithms of the 2D algorithms
2. Develop 1D/2D algorithms using common functions in the utilities library
3. Change input data types
4. Direct translate to native code on changes above, allowing reduced delay in runtime
5. Catch possible bugs immediately if changes are faulty

For testing purposes, we have assembled all 1D and 2D algorithms in `1d_algo_set_main.cpp` and `2d_algo_set_main.cpp` respectively. In one dimensional algorithms, the input consists of a file with array length in the 1st row. Matrices are represented with row length in the 1st line and column length in the 2nd line. The rest of the lines are values of the matrices that is obtained by scanning matrix from left to right, row by row.

To make this library to compile without linking, no explicit routines implemented. All methods and utilities are in header files.

4.2 Using the Library

All 2D partitioning classes take following template parameters:

- **T**: Type of data.
- **Pr**: A matrix data structure that will hold prefix sum data.

`Pr` can be defined by developer but it needs to overload `[]` operator and implements `sizeX` and `sizeY` methods. `sizeX` and `sizeY` must return 1 more than original matrix size because 1st row and first column values are all zero. Therefore `Pr[0][y]` `Pr[x][0]` must return 0 where `x<sizeX` and `x<sizeY`. For example a data structure that uses integer arrays can be defined as in Code 4.1.

It is important to note that even though `Pr` is used to represent 1D arrays too, it has nothing to do with matrix `Pr`. One dimensional `Pr`, which can be seen in 1D partitioning algorithms, do not have to implement `sizeX` or `sizeY` methods. `[]` overload is sufficient.

A user can run Nicol's algorithm by using lines in Code 4.2.

Code 4.1: Initializing prefix sum array

```
#include "util/Compact.hpp"
#include "util/Prefix2D.hpp"

typedef int T;
typedef Prefix2D<T> Pr;
Compact2D<T> data(sizeX, sizeY); //creates a matrix
readFromFile(data) //puts file contents into matrix
Pr pr(sizeX, sizeY, data); //converts matrix into prefix sum matrix
```

Code 4.2: Running the algorithm

```
#include "twod/part_base.hpp"
#include "util/rect_list.hpp"
#include "twod/parse.hpp"

//Create an instance of part. alg. object
twod::PartBase<T, Pr >* pb = twod::parseName<T,Pr>("RECT-NICOL");
//Create a rectangle list to hold the results
Rect_list<T, Pr > rl(pr);
//Bottleneck value goes to b variable and rl
//contains partition rectangles
T b = pb->part(procCount, pr, rl);
delete pb;
```

Code 4.3: Rectangle structure members

```
struct rectangle
{
  /// coordinate x - rectangle top left
  int x_top_l;
  /// coordinate y - rectangle top left
  int y_top_l;
  /// coordinate x - rectangle bottom right
  int x_bot_r;
  /// coordinate y - rectangle bottom right
  int y_bot_r;
}
```

Code 4.4: Writing result to standard output

```
util::Rect_list<T, Pr>::container::const_iterator list_iter1 = rl.
    rectangles.begin();//define iteratior
//show reult one rectangle per line format
for(;list_iter1 != rl.rectangles.end(); list_iter1++)
  {
    const rectangle &r =  *list_iter1;
    std::cout << r.x_top_l  <<  "␣" << r.y_top_l << "␣";
    std::cout << r.x_bot_r  <<  "␣" << r.y_bot_r << std::endl;
  }
```

The algorithm type can be easily changed by using proper class name. string Now `rl` contains a list of rectangles. `rectangle` structure is defined in `util/rect_list.hpp`. The members of rectangle structure are given in Code 4.3.

To get the rectangles that represent the partition, one can obtain an iterator and iterate through `rectangles` member of `Rect_list` class as seen in Code 4.4.

4.3 One Dimensional Partitioning Implementation Details

Even though 1D partitioning algorithms are mainly used in 2D partitioning algorithms, it is also possible to use them independently. In the testing code, a function pointer is assigned the proper function based on the command line argument. The

total number of processors and file name are also passed as a command line argument by the user. Using those information, the desired 1D partitioning function is called on a load array which is created by input file. The function returns the maximum loaded processor's load and another pointer (which is passed as a parameter) retains the cut points.

Each cut point maps to an index of load array. There are $m-1$ cut points. Given a set of processors $\{p_1, p_2, \ldots p_m\}$ and cut points $\{c_1, c_2, \ldots c_{m-1}\}$ The load in the index of c_i belongs to p_i. For instance array an with values 3, 2, 3, 1, 5 would have a prefix sum array 0, 2, 5, 8, 9, 14. A cut point positioned in index 3 would divide array into 2 processors. The processors would have load of 8 and 6 respectively. The following sections discuss one dimensional algorithms' implementations. Even though the function bodies seem different, each function is wrapped by a single function(Code 4.5) to let programmers easily plug different algorithms as needed. For example `JAG-M-HEUR` uses Nicol's 1D partitioning algorithm. One can easily exchange underlying algorithm by changing a few lines (feature 1). Binary search is used by all algorithms to search cut points given a bottleneck value. This is kept in a common library (Feature 2).

4.3.1 `DirectCut`

Code 4.6 is the simplest one dimensional partitioning algorithm. This implementation assumes that the input array consists of nonzero values. `val` variable keeps track of maximum sum of bottleneck values until processor p (including `p`). `avg` variable holds the best bottleneck value possible. `(p+1) * val` must give us the maximum load between array index 0 to pth cut. To limit binary search region, binary search made between `step` and `step - inc`. The length of `inc` which is step length is arbitrary. Too small step length will cause too many, but short binary search bursts.

Code 4.5: General one dimensional partitioning interface

```
/**
 * General 1d partitioning header
 *
 * @param procCount Total number of processors
 * @param prefixSumArray Prefix sum array that always starts with
    value 0 in the 0th index
 * @param length length of prefixSumArray
 * @param cutIndexes Cut index points. A cut index point p is
    inclusive to processor p-1 exclusive to processor p
 * @param max Maximum element in the actual array (this value is
    considered only by Nicol's 1D partitioning algorithm use -1 if
    you don't know what it is)
 *
 * @return
 */
static T part(int procCount, const Pr& prefixSumArray,
              int length, int *cutIndexes, T max);
```

On the other hand, longer step lengths will cause fewer but longer binary searches. The step length can vary but in order to preserve runtime complexity in [29] `length / procCount` is chosen.

`DirectCut` with refined bottleneck

This is another variance of `DirectCut` that recalculates `avg` value every time a cut placed. The rest of the operations are the same as `DirectCut` (Code 4.7).

4.3.2 `NicolPlus`

`NicolPlus` is the fastest optimum 1D partitioning algorithm. Bottleneck lower bound is used to restrict search space of `NicolPlus`, to speedup calculations. Bottleneck lower bound is initialized to average load. The most important feature of this algorithm is that it keeps track of index lower and upper bounds in two separate array for each processor. This property significantly decreases binary search times in each step. Just like bottleneck lower and upper bounds, index lower and upper bounds

Code 4.6: Direct cut algorithm

```
/**
 * Applies direct cut algorithm to a given prefixSumArray
 * @param procCount is the total number of processors
 * @param prefixSumArray always begins with 0 as first element.
 * @param length is the exact size of prefixSumArray
 * @param cutIndexes array of cuts
 * @param T data type of 1d array
 *
 * @return *cutIndexes (must be allocated before calling!)
 */
static T direct_cut(int procCount, const Pr& prefixSumArray, int
   length, int *cutIndexes, T)
{
  T avg = prefixSumArray[length - 1] / procCount;
  int inc = length / procCount, p, lastcut = 0;
  int currentcut;
  int step = inc;
  T bottleneck = -1, val;
  for(p = 0; p < procCount - 1; p++)
    {
      val = (p + 1) * avg;
      while(prefixSumArray[step] <= val)
        {
          step+=inc;
          if(step > length-1)
            step = length-1;
        }
      currentcut =  binarySearchLeft(prefixSumArray, step - inc,
         step, val);
      if (cutIndexes != NULL)
        cutIndexes[p] = currentcut;
      if((prefixSumArray[currentcut] - prefixSumArray[lastcut]) >
         bottleneck)
        bottleneck = prefixSumArray[currentcut] - prefixSumArray[
           lastcut] ;
      lastcut = currentcut;
    }
      if(prefixSumArray[length - 1] - prefixSumArray[lastcut] >
         bottleneck)
        return prefixSumArray[length - 1] - prefixSumArray[lastcut];
  else
    return bottleneck;
}
```

Code 4.7: Direct cut with refined bottleneck

```
for(p = 0; p < procCount - 1; p++)
   {
     avg = (prefixSumArray[length-1] - prefixSumArray[lastcut])/ (
        procCount - p);
     //Followed by the same operations as DirectCut
   }
```

Code 4.8: Nicol's 1D partitioning algorithm

```
 /**
  * @param procCount number of processors
  * @param prefixSumArray Prefix sum array that always starts with
     value 0 in the 0th index
  * @param length is the length of w
  * @param cutIndexes cut index points
  * @param max maximum load of a given array. If unknown, use -1.
  * @param *sl holds lower index bounds of each processor
  * @param *sh holds higher index bounds of each processor
  * @return bottleneck value and *cutIndexes
  */
 static T nicol_plus(int procCount, const Pr & prefixSumArray,
                     int length, int *cutIndexes, T max,  int *sl,
                        int *sh)
```

are updated in every iteration. Basically, `NicolPlus` applies a binary search over bottleneck value and returns the least feasible bottleneck.

`RProbe`

`RProbe` (Code 4.9) simply goes through every processor and greedily maps tasks to processors. It applies binary search in code to find such an index.

4.3.3 `Recursive Bisection`

Recursive bisection decides the cut point by minimizing load difference between left and right sides of the cut point. Pure recursive bisection may not be efficient when

Code 4.9: Testing Bottleneck feasibility with RProbe

```
//returns 1 if it is possible to partition an array with a given
//bottleneck value B else 0. b is the base index
static int rprobe(const Pr& wpre, int length, const T& bottleneck,
                  int numproc,  int *s, int *sl, int *sh)
{
  T bsum = bottleneck;
  int p;
  for(p = 0; p < numproc - 1; p++)
    {
      s[p] = binarySearch(wpre, sl[p], sh[p], bsum);
      bsum = wpre[s[p]] + bottleneck;
    }
  if(bsum >= wpre[length - 1])
    return 1;
  else
    return 0;
}
```

processor count is odd. To overcome this issue, the load of each side is normalized to the processors number assigned. Code 4.11 shows how this calculation is done.

Code 4.12 finds the minimum cut point by checking the load. When deciding in the middle cut point, it tries the both middle and middle+1 index and selects the minimum one. The reason to do that is avoid misplaced cuts due to integer division.

4.3.4 Calculating Lower and Upper Bounds

Limiting the bottleneck search space can lead to faster result acquisition. This is especially obvious in Nicol's 1D partitioning algorithm. In all 1D partitioning algorithms those bounds are used. The upper bound corresponds to `DirectCut` proof in section 2.1.2 and the lower bound corresponds to L_{avg}.

Code 4.10: Recursive Bisection

```
 /**
   * Performs 1D recursive bisection algorithm
   *
   * @param procCount total number of processors
   * @param prefixSumArray prefixSumArray
   * @param length length of prefixSumArray
   * @param cutIndexes cut point results
   *
   * @return *cutIndexes and bottleneck
   */
  static T rec_bisection(int procCount, const Pr& prefixSumArray,
                         int length, int *cutIndexes, T)
```

Code 4.11: Normalized load calculation

```
/**
 * Calculates the normalized maximum load given a  cut point and
    processors for each side
 * @param prefixSum prefix sum array
 * @param low subPrfixSumArray's leftmost cut point
 * @param high subPrfixSumArray's rightmost cut point
 * @param leftProc processor count to be assigned to candidate's cut
    point's left
 * @param rightProc processor count to be assigned to candidate's
    cut point's right
 * @param candidate a cut point (low<=candidate<=high)
 *
 * @return maximum load given cutpoint and subarray
 */
double calculateDiff(const Pr& prefixSum, int low, int high, int
   leftProc, int rightProc, int candidate)
{
  double leftLoad = (prefixSum[candidate] - prefixSum[low - 1]) /
     leftProc;
  double rightLoad = (prefixSum[high] - prefixSum[candidate])/
     rightProc;
  return max(leftLoad,rightLoad);
}
```

Code 4.12: Finding even cut point

```
  /**
   * Finds the best cut point of a given array by calculating
   * Wtot*(leftProc)/(leftProc+rightProc) and then decides
   * whether to cut from left or right.
   * Left and low index values are included to
   * calculation. prefixsum[0] = 0.
   *
   * @param prefixSum 1D prefixsum array
   * @param low lower bound
   * @param high upper bound
   * @param leftProc number of processor on left side
   * @param rightProc number of processor on right side
   *
   * @return even cut index
   */
static int findEvenCut(const Pr& prefixSum, int low, int high,
                       int leftProc, int rightProc)
{
  T cutWeight = (T) ((prefixSum[high] - prefixSum[low - 1]) * (
     double)leftProc / ((double)leftProc + (double)rightProc));
  int cutPoint = binarySearch(prefixSum, low, high, cutWeight+
     prefixSum[low-1]);
  if (cutPoint == -1)
    return low;
  double d1 = calculateDiff (prefixSum, low, high, leftProc,
     rightProc, cutPoint);
  double d2 = calculateDiff (prefixSum, low, high, leftProc,
     rightProc, cutPoint+1);
  return d1 < d2 ? cutPoint  : cutPoint+1;
}
```

4.4 Two Dimensional Partitioning Implementation Details

The library aimed to give immediate feedback to false manipulations. Just like 1D partitioning algorithms, both inline assertions, test cases and regression tests are provided for developers so that they can improve the code without fear of causing errors (Feature 5). 2D algorithms can be more complex due to special data structures they use such as sub-matrices. Those data structures are no exception to inline assertions so a developer can focus on algorithm itself rather than worrying about data structures. All two dimensional algorithms share common properties. Those are:

- Using a prefix sum array as input
- Using a rectangle list as output (not necessarily ordered)
- Having nonzero elements

Matrix transpose is a common operation in all 2D partitioning algorithms. This is due to the fact that cut orientations (horizontal or vertical) may change the quality of partitioning. To reduce code duplication, `TransposePrefix2D` is used. This class simply reverses the coordinates and returns the value. This can be accomplished by overloading `[]` operator. The important parts of `TransposePrefix2D` is given in Code 4.13. The purpose of `Intermediate` inner-class is to save the first index number.

4.4.1 Reducing a Matrix into an array

The most challenging part of developing 2D algorithms is dynamically building set of 1D arrays and modifying the 1D algorithm to consider set of arrays every time a bottleneck query made. Those sets are determined by previous' dimensions cut points.

Code 4.13: `TransposePrefix2D` class

```
template <typename T, typename Pr>
class TransposePrefix2D
{
public:
  Intermediate operator[] (int x) const
  {
    return Intermediate(*this,x);
  }
  class Intermediate
  {
  public:
    const TransposePrefix2D& a;
    int x;
    Intermediate(const TransposePrefix2D& ar, int xval)
      :a(ar),x(xval){}
    T operator[] (int y) const
    {
      return a.originalMatrix[y][x];
    }
  };
}
```

To represent those sets of arrays, we must first find a way to build chunks of arrays given cut points in the previous dimension. `AggregMax2Dto1D` class (Code 4.14) is used for this purpose. A similar class, `Aggreg2Dto1D`, is used by the other 2D algorithms. (see section 4.4.7). Therefore it resides in `util` namespace (Feature 2).

4.4.2 `PartBase` class

All 2D partitioning classes derive from this abstract class so as to provide single interface to all 2D partitioning algorithms. This will allow us to use polymorphism as needed.

4.4.3 `RECT-UNIFORM`

Divides geometric space regardless of load. It is the fastest, yet the simplest algorithm in load imbalance.

Code 4.14: `Aggreg2Dto1D` class

```
/**
 * Generates 1D representation of a subsection of a prefix sum array
 *
 * This is a 1d prefix sum array with both [] notation and interval
 * notation.
 *
 * @param T data type of the array
 * @param Pr type of the container array
 * @param row select the orientation of the summation
 * If row is true and the rectangle is of size 10x20, then there
 * equivalent 1darray is of size 10.
 */

template <typename T, typename Pr, bool row>
class Aggreg2Dto1D;
 /**
  * Reduces the 2D matrix into 1D array \f$[xl:xh]\times[yl:yh]\f$
     in the
  * original array \f$[1:n]\times[1:m]\f$ (where n and m are last
     elements
  * not sizes).
  *
  * If row is true, there are (xh-xl) elements in the array.
  *
  * @param p prefixSumArray
  * @param xl xlow of submatrix
  * @param xh xhigh of submatrix
  * @param yl ylow of submatrix
  * @param yh yhigh of submatrix
  */
Aggreg2Dto1D(const Pr& p, int xl, int xh, int yl, int yh);
```

Code 4.15: `AggregMax2Dto1D` class

```
/**
 * Goes through all the 1st Dimension fixed rows and returns the
    most
 * loaded row interval over all rows
 *
 * @param left lower interval bound index
 * @param right upper interval bound index
 *
 * @return
 */

T interval(int left, int right) const
{
  T maxVal;
  int i = 0;
  while (lineSet[i] == NULL)
    {
      assert (i < lineCount);
      i++;
    }
  maxVal = lineSet[i]->interval(left,right);

  for(; i < lineCount; i++)
    {
      if (lineSet[i] != NULL)
        maxVal = std::max(lineSet[i]->interval(left,right), maxVal);
    }
  return maxVal;
}
```

Code 4.16: HIER-RB options to set cut orientation

```
///Cut to minimize load
const static int LOAD=0;
///Cut to minimize side-to-side distance
const static int DIST=1;
///start vertically and alternate between horizontal
const static int VERT_ALT=2;
///start horizontally and alternate between vertical
const static int HOR_ALT=3;
template<typename T, typename Pr,  int type=0>
class Rec_bisect_2d : public PartBase<T,Pr>
```

4.4.4 RECT-NICOL

The main idea of this algorithm is to iteratively refine one dimension by fixing the other dimension. This allows us to use any 1D algorithm. The Algorithm, just like others, allows replacing underlying one dimensional algorithm with another one (Feature 1) It uses `AggregMax2Dto1D` class for row reduction.

4.4.5 HIER-RB

Recursive bisection is implemented with different variations based on initial cut orientation as described in section 2.4.3. Code 4.16 shows the options.

4.4.6 HIER-RELAXED

`HIER-RELAXED` has a modification in the function that finds the even cut. Relaxed version tries all possible cut points within the given sub-rectangle and selects the best one. As described in section 2.3.2, the bottleneck/cut point function is bi-monotonic. To speedup search in this function, a bi-monotonic binary search method is implemented. The important aspect of this binary search is that it decides bounds by checking the slope. Code 4.17 shows how bounds refined.

Code 4.17: Bound refinement in bi-monotonic binary search

```
if(slopeAt(prefixSum, low, high,  leftProc, rightProc,  mid) == -1)
    lowlimit = mid+1;
 else
   highlimit = mid;

/**
 * @param prefixSum 1D prefix sum array
 * @param low Lowest index limit of prefix sub-array
 * @param high Highest index limit of prefix sub-array
 * @param leftProc number of procs to be assigned to the left
 * @param rightProc number of procs to be assigned to the right
 * @param point The index of cut point that we want to retrieve
    Bottleneck/Cut point sign
 *
 * @return
 * Returns a signature
 * -1 means we are in decreasing side of the function
 * +1 means we are in increasing side of the function
 */
static int slopeAt(const Pr prefixSum, int low, int high, int
   leftProc, int rightProc, int point);
```

4.4.7 JAG-PQ-HEUR,JAG-M-HEUR and JAG-M-PROBE

All jagged algorithms use `Aggreg2dto1D` class to represent row chunks. This class implements row chunk notation for 1D array. The data comes from a 2D prefix sum array. The size of the exposed 1D array is given by the first dimension of the 2D prefix sum array. When ask for the value of an interval (in the first dimension), the stripe defined by the interval is returned. JAG-PQ-HEUR algorithm has variances that depends on first dimension choice. Load-based variance tries both dimensions as first dimension and returns the best one.

4.4.8 JAG-PQ-OPT and JAG-M-OPT

Both `JAG-PQ-OPT` and `JAG-M-OPT` are dynamic programming algorithms. In our experiments, we have seen that both algorithms use vast amount of memory so we

Code 4.18: JAG-PQ-HEUR class with variations horizontal-first, vertical-first, and best-load variances

```
/**
 * @brief implements a PxQ jagged partitioning heuristic using the
 * first dimension as main dimension.
 *
 * @param T data type of instance matrix
 * @param Pr data type of 2D matrix
 * @param *onedalgoY algorithm to be used to fix columns
 * @param *onedalgoX algorithm to be used to fix rows
 */
template <typename T, typename Pr,
          T (*onedalgoY) (int procCount, const util::Aggreg2Dto1D<
              T, Pr, false>& prefixSumArray, int length, int *
              cutIndexes, T) = oned::NicolPlus<T, util::
              Aggreg2Dto1D<T, Pr, false> >::nicol_plus,
          T (*onedalgoX) (int procCount, const util::Aggreg2Dto1D<
              T, Pr, true>& prefixSumArray, int length, int *
              cutIndexes, T) = oned::NicolPlus<T, util::
              Aggreg2Dto1D<T, Pr,  true> >::nicol_plus >
class Her_jag_2d_horf : public PartBase<T,Pr>
class Her_jag_2d_verf : public PartBase<T,Pr>
class Her_jag_2d_best : public PartBase<T,Pr>
```

focused on maximizing cache locality and minimizing memory access. There are 3 parameters to memorize those are 1D array length, thickness and number of processors in this array. As you can see, this needs a 3D data structure to hold. To achieve locality with 3D array, (which is required to save the result of a 1D partitioning). `Compact3D` class is implemented. This class saves the result in 1D array. The advantage of this approach comes from allocating single array instead of multiple arrays and avoiding pointer array traverses.Another optimization aims to reduce calls to 1D algorithm. We do that by calculating upper and lower bounds. Code 4.19 show bounding methods. Line 9 checks whether a calculation is occurred in the past (non-negative value). If calculation was made, it is fetched and returned. Otherwise, a chunk of rows are created in 11. This chunk n rows. Obviously the best load is

the total load of the chunk divided by m. Upper bound calculation is carried on the same way except the total load of the chunk returned this time.

4.5 Extending the Library

In this section, we will focus on tips to extending and modifying 2D partitioning library. As explained earlier, all 2D partitioning algorithms use a 1D partitioning algorithm. A developer may choose one of 1D partitioning algorithms in the library, or develop his/her own. After that, 1D algorithm can be plugged in one of the existing 2D algorithms. If a new 2D algorithm is desired, then it must inherit from `PartBase` class. We will go through a simple toy example to modify a 2D algorithm.

Let us try uniform 1D algorithm in the 1st dimension to speedup partitioning process. Uniform 1D simply divides array into P equal intervals. It is completely insensitive to load so that it runs very fast. Since uniform class is already in the library, we plug it (`oned/uniform.hpp` should be included in Code 4.20).

If 2D algorithm modification is required rather than 1D, developer has to

1. Inherit from `PartBase` class thus override `part()` method. `part()` returns bottleneck value.

2. Save results in `Rect_list` class which take rectangle as element.

It is important to test validity of resulting partition. Developers may use `bool Rect_list::valid_part()` method whenever needed. `Aggreg2dto1D` class is another very useful data structure that reduces any 2D sub-rectangle into 1D array by summing elements up in desired order. We set sum order by changing template parameter `row`. If `row` is set to true, the sub-rectangle is reduced to first dimension by summing elements. Otherwise, the it is reduced to 2nd dimension.

Code 4.19: Lower and upper bound calculation in dynamic programming

```
1  /**
2   * @brief return a lower bound on the best partitioning of [1:n]
3   * on m processors using m-way jagged partitioning
4   */
5  T algo_2d_lb(int n, int m)
6  {
7    if (n == 0) return 0;
8    if (m == 0) return infinite();
9    if (dp_array[n][m] < 0)
10     {
11       util::Aggreg2Dto1D<T, Pr, false> ps1dx_row(psa, 1, n, 1, psa.
             prefixsizeY() - 1);
12       return ps1dx_row[psa.prefixsizeY()-1]/m;
13     }
14   else
15     return dp_array[n][m];
16 }
17
18 /**
19  * @brief return an upper bound bound on the best partitioning of
20  * [1:n] on m processors
21  */
22 T algo_2d_ub(int n, int m)
23 {
24   if (n == 0) return 0;
25   if (m == 0) return infinite();
26   if (dp_array[n][m] < 0)
27     {
28       util::Aggreg2Dto1D<T, Pr, false> ps1dx_row(psa, 1, n, 1, psa.
             prefixsizeY() - 1);
29       return ps1dx_row[psa.prefixsizeY()-1];
30     }
31   else
32     return dp_array[n][m];
33 }
```

Code 4.20: Plugging another algorithm in the 1st dimension

```
typedef long int T;
typedef Prefix2D<T> Pr;
Compact2D<T> data(sizeX, sizeY);
readFromFile(data)//Load data to matrix
Pr pr(sizeX, sizeY, data);//Convert matrix to prefix sum array
Rect_list<T, Pr > rl(pr);//Prepare rectangle list to save results
twod::PartBase<T, Pr >* pb = new twod::M_way_probe_horf<T, Pr, false
    ,
oned::Uniform<T, util::Aggreg2Dto1D<long int, Pr, false> >::uni_cut,
oned::NicolPlus<T, util::Aggreg2Dto1D<long int, Pr, false> >::
    nicol_plus> ();//Create algorithm instance
T d = pb->part(procCount, pr, rl);//run Algoritm
```

It is recommended to check out existing helper utilities before writing a new one. For instance, dividing an array into two equal parts is commonly used operation in bisection algorithms. If a developer comes up with an idea that requires to do so, he/she can use `oned::RecursiveBisection::findEvenCut()`.

Zero loaded tasks can create special cases. Developers should be aware of those conditions and write test cases accordingly. For example zero tasks can leave one or more processors empty. In this case it is not a good idea to assume output rectangle count will be the same as input processor number.

CHAPTER 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 non-negative integers 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-PROBE` and `HIER-RELAXED` heuristics we proposed get significantly better load balances than existing algorithms while running in less than a second. We showed how `HYBRID` algorithms can be engineered to achieve better load balance but use significantly more computing time. Finally, if computing time is not really a limitation, one can use more complex algorithm such that `JAG-M-OPT`.

Showing that the optimal solution for m-way jagged partitions, hierarchical bipartitions and hierarchical k-partitions with constant k can be computed in polynomial time is a strong theoretical result. However, the runtime complexity of the proposed dynamic programming algorithm remains high. Reducing the polynomial order of these algorithms will certainly be of practical interest.

We also showed that `JAG-M-HEUR-PROBE` is better than rectilinear algorithms in terms of total communication cost but rectilinear algorithms are more acceptable

when network contention is on one node. Dynamic application will require rebalancing and the partitioning algorithm should take into account data migration cost. Hierarchical algorithms degrade slower than other classes of solutions therefore they can be used if the task relocation is chaotic.

We only considered computations located in a two dimensional field but some applications, such as PIC-MAG and SLAC, might expose three or more dimensions. A simple way of dealing with higher dimension would be to project the space in two dimensions and using a two dimensional partitioning algorithm, as we have done in some of the applications. But this choice is likely to be suboptimal since it drastically restrict the set of possible allocations. An alternative would be to extend the classes of partitions and algorithm to higher dimension. For instance, a jagged partitioning algorithm would partition the space along one dimension and perform a projection to obtain planes which will be partitioned in stripes and projected to one dimensional arrays partitioned in intervals. All the presented algorithms extend in more than two dimensions, therefore the problems will stay in the same complexity class. However, the guaranteed approximation is likely to worsen, the time complexity is likely to increase (especially for dynamic programming based algorithms). Memory occupation is also likely to become an issue and providing cache efficient algorithm should be investigated. However, the increase of the size of the solution space will provide better load balance than partitioning the two dimensional projection.

We are also planning to integrate the proposed algorithms in a distributed particle in cell simulation code. To optimize the application performance, we will need to take into account communication into account while partitioning the task. Finally, to keep the rebalancing time as low as possible, it might useful not to gather the load information on one machine but to perform the repartitioning using a distributed algorithm.

BIBLIOGRAPHY

[1] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proc. of IPDPS*, 2004.

[2] M. Aftosmis, M. Berger, and S. Murman. Applications of space filling curves to cartesian methods for CFD. In *Proc. of the 42nd AIAA Aerospace Sciences Meeting*, 2004.

[3] S. Aluru and F. E. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proc. of the 4th IEEE Conference on High Performance Computing*, pages 230–235, 1997.

[4] B. Aspvall, M. M. Halldórsson, and F. Manne. Approximations for the general block distribution of a matrix. *Theoretical Computer Science*, 262(1-2):145–160, 2001.

[5] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C36(5):570–580, 1987.

[6] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, 1988.

[7] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[8] U. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999.

[9] U. V. Çatalyürek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Fisk. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, 2009.

[10] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. *Journal of Parallel and Distributed Computing*, 47:139–152, 1997.

[11] H. P. F. Forum. High performance FORTRAN language specification, version 2.0. Technical Report CRPC-TR92225, CRPC, Jan. 1997.

[12] G. N. Frederickson. Optimal algorithms for partitioning trees and locating p-centers in trees. Technical Report CSD-TR-1029, Purdue University, 1990, revised 1992.

[13] D. R. Gaur, T. Ibaraki, and R. Krishnamurti. Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem. *Journal of Algorithms*, 43(1):138–152, 2002.

[14] A. Grama, G. Karypis, V. Kumar, and A. Gupta. In *Introduction to parallel computing*, chapter 2. Addison-Wesley, 2nd edition, 2003.

[15] M. Grigni and F. Manne. On the complexity of the generalized block distribution. In *Proc. of IRREGULAR '96*, pages 319–326, 1996.

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

[17] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 2000.

[18] V. Horak and P. Gruber. Parallel numerical solution of 2D heat equation. In *Parallel Numerics '05*, pages 47–56, 2005.

[19] H. Karimabadi, H. X. Vu, D. Krauss-Varban, and Y. Omelchenko. Global hybrid simulations of the earth's magnetosphere. *Numerical Modeling of Space Plasma Flows*, Dec. 2006.

[20] G. Karypis and V. Kumar. *MeTiS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[21] G. Karypis, V. Kumar, R. Aggarwal, and S. Shekhar. *hMeTiS A Hypergraph Partitioning Package Version 1.0.1*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[22] S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangle tiling and packaging. In *Proc. of the 19th SODA*, pages 384–393, 1998.

[23] S. Khanna, S. Muthukrishnan, and S. Skiena. Efficient array partitioning. In *Proc. of ICALP '97*, pages 616–626, 1997.

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

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

[26] J. Y.-T. Leung. Some basic scheduling algorithms. In J. Y.-T. Leung, editor, *Handbook of Scheduling*, chapter 3. CRC Press, 2004.

[27] F. Manne and B. Olstad. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[28] F. Manne and T. Sørevik. Partitioning an array onto a mesh of processors. In *Proc of PARA '96*, pages 467–477, 1996.

[29] 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*, pages 550–564, 1997.

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

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

[32] K. Paluch. A new approximation algorithm for multidimensional rectangle tiling. In *Proc. of ISAAC*, 2006.

[33] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, 1996.

[34] A. Pınar and C. Aykanat. Sparse matrix decomposition with optimal load balancing. In *Proc. of HiPC 1997*, 1997.

[35] A. Pınar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64:974–996, 2004.

[36] A. Pınar, E. Tabak, and C. Aykanat. One-dimensional partitioning for heterogeneous systems: Theory and practice. *Journal of Parallel and Distributed Computing*, 68:1473–1486, 2008.

[37] 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*, 152(3):227 – 241, 2003.

[38] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Euro-Par*, pages 296–310, 2000.

[39] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. of SuperComputing '00*, November 2000.

[40] M. Ujaldon, S. Sharma, E. Zapata, and J. Saltz. Experimental evaluation of efficient sparse matrix distributions. In *Proc. of SuperComputing'96*, 1996.

[41] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.