© 2015 Laura Richardson

THE NON-LINEAR PARALLEL OPTIMIZATION TOOL (NLPAROPT): A SOFTWARE OVERVIEW AND EFFICIENCY ANALYSIS

BY

LAURA RICHARDSON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Victoria Coverstone

ABSTRACT

Modern spacecraft trajectory mission planning regularly involves some component of numerical optimization to provide a significant improvement in a solution, and in some instances is the only manner for finding an acceptable solution to a problem. One of the most useful constrained optimization formulations is the Non-Linear Programming (NLP) problem. NLP problem solvers have been used to analyze complex optimization problems by space mission designers as well as a wide variety of other fields such as economics and city planning.

There are several state-of-the-art software packages that are currently used to solve NLPs in the aerospace industry, such as SNPOT, IPOPT, and WORHP. Most of the critical space trajectory design software tools in use today, such as OTIS, EMTG, and MALTO, are dependent on a drop-in NLP solver (typically SNOPT or IPOPT); this dependence is not unwarranted, as the existing solvers are especially adept at solving very large-scale, preferably sparse, non-linear programming problems. However, neither SNOPT nor any other NLP solver available exploits parallel programming techniques. Most trajectory design tools have already started to implement parallel techniques, and it is now the serial-execution nature of the existing packages that represents the largest bottleneck. A high-fidelity trajectory optimization NLP can take days, weeks, or even months before converging to a solution, resulting in a prohibitively expensive mission planning design tool, both in terms of time and cost.

CU Aerospace in partnership with the University of Illinois at Urbana-Champaign (UIUC) has developed a novel, ground-up redesign NLP solver that takes advantage of high performance parallel computing called the Non-Linear PARallel Optimization Tool (NLPAROPT). NLPAROPT utilizes the Boost.MPI library, a message-passing interface that allows for point-to-point communication among the processors, as well as the Linear Algebra PACKAGE (LAPACK) and the Basic Linear Algebra Subprogram (BLAS) functions to carry out complex linear algebra calculations in a parallel fashion. Preliminary tests have shown NLPAROPT's ability to reduce the runtime by orders of magnitude when compared to its serial counterpart. A full discussion of NLPAROPT's speed-up results can be found in the SBIR Phase I final report [1].

The primary topics of this thesis are a software overview followed by an in-depth analysis of NLPAROPT's efficiency. A trade-off analysis was conducted to determine the optimal number of processors to run a given problem with a certain computational complexity measured in Floating Point OPerations (FLOPs). Additionally, a series of algorithms were developed in MATLAB to output the optimal number of two-dimensional block-cyclic blocking factors to ensure the processor idling time is minimized, thereby minimizing the overall runtime of the program. The results of using optimal versus non-optimal blocking factors have proven the processor idling time is significantly reduced. Based on these analyses, any astrodynamics researcher can use the trends from these results to efficiently run any optimization problem in NLPAROPT.

NLPAROPT already shows great promise as a rapid, robust NLP solver. Unexplored parallelization opportunities still exist within NLPAROPT. After these parallelization techniques have been implemented and the core algorithm is further refined, NLPAROPT will be transparently usable by the existing optimal trajectory solvers used by astrodynamicists, scientists, and researchers of all kinds across the globe.

ACKNOWLEDGEMENTS

To my graduate advisor, Dr. Victoria Coverstone: Thank you for always being a kind, patient, motivational, lively, spunky, and insightful advisor the past two years. I am grateful for your constant support, especially when I wanted to change my thesis topic three times.

To Dr. Alex Ghosh and the rest of the NLPAROPT Phase I team: Thank you for making me constantly work hard and dive into unfamiliar territory to learn parallel programming. Developing NLPAROPT has been one of my fondest and proudest memories, and I am forever grateful for the opportunity to work with all of you.

To Dr. David Carroll: Thank you for helping me find my way as a young, budding engineer and for being an invaluable professional mentor and friend throughout my undergraduate and graduate years.

To Darren King and Curtis Woodruff: Thank you for the frequent breakfast chats at Merry Ann's. A tradition so small turned into something that undoubtedly kept me sane throughout the last several years.

To the brilliant engineers at CU Aerospace: Thank you for constantly giving me the opportunity to learn, and also for serving as professional (and oftentimes personal) mentors.

To Tom: Thank you for being my emotional rock during moments of complete stress and sleep-deprivation. There were many times you convinced me not to quit, and I am so glad you were there to point me in the right direction.

To my parents: Thank you for financially and emotionally supporting my seven years of college. Most importantly, thank you for teaching me that the most important things in life can't be learned by memorizing an orbital mechanics textbook (no offense to Prussing and Conway), but rather by travelling the world and gaining a love and appreciation for all of the wonders it has to offer, especially the people who live in it.

The Small Business Innovative Research (SBIR) Phase I effort to research and develop NLPAROPT was funded by NASA with Dr. Jacob Englander as the Contract Officer Technical Representative from the NASA Goddard Space Flight Center.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Non-Linear Programming problems (NLPs) exist in many different research and industry areas, including all engineering fields, business, and city planning and zoning. The goal of the NLP is to optimize (minimize or maximize) defined parameters given certain non-linear constraints; the applications of such a problem formulation are practically endless. Though robust, state-of-the-art serial NLP solvers currently exist and are frequently utilized by the spacecraft trajectory optimization and mission planning groups at NASA, Boeing, and universities across the world, one major problem with these solvers is the prohibitive nature of large-scale optimization problems. The computationally complex problems that these groups are often tasked with solving may have hundreds to thousands of variables and constraints. Even the most intelligent (and bored) graduate student would find solving such an NLP by hand to be an impossible task, which is why the computer NLP solvers have been an important addition to the optimization community. However, though these solvers can accurately solve large-scale NLPs, the runtime required to converge to a solution can take hours, weeks, or even months, depending on the size of the problem. The incredibly large computational runtimes can prove costly and also prohibit the project team from progressing in their design process while waiting for a solution. The engineering community especially has expressed the need for a parallelized NLP solver; rather than running the solver on one computer in a designated serialized fashion, the computations are split among multiple processors to significantly speed up the solution process.

In June 2014, CU Aerospace (CUA) was awarded a NASA Small Business Innovative Research (SBIR) Phase I contract to research and develop a Non-Linear PARallel OPtimization Tool (NLPAROPT). Together with a research team at the University of Illinois at Urbana-Champaign (UIUC), a prototype NLPAROPT was developed and delivered to the NASA Goddard Space Flight Center in December 2014. The prototype incorporated parallel computing routines utilizing Boost.MPI (a message passing interface that allows for point-to-point and collective communication among all of the processors) and CUDA (a GPU-based parallel computing platform). Several test cases were run against the serialized NLPAROPT and significant speed-ups for a wide variety of test cases were demonstrated. Throughout the Phase I effort, however, an efficiency analysis was never conducted on NLPAROPT to determine if any inefficiencies exist that would inhibit the minimization of the total computational runtime; conducting such an analysis

and determining the exact inefficiencies that exist and how they can be mitigated would allow for a fully robust parallelized optimization program. This thesis outlines the research and algorithm development as part of the Phase I effort, as well as an in-depth efficiency analysis and an outline of the future work that the CUA and UIUC team is eager to explore.

1.1 NON-LINEAR PROGRAM (NLP)

An NLP is an optimization problem that contains a non-linear objective function and/or constraints and can be written in the following format:

$$NLP = \begin{cases} \min\limits_{x \in D \subseteq \mathbb{R}^n} f(x)\colon\ \mathbb{R}^n \to \mathbb{R} \\ subject\ to \quad l \leq \begin{pmatrix} x \\ Ax \\ c(x) \end{pmatrix} \leq u \end{cases} \tag{1}$$

In Equation (1), $f(x)$ represents the objective function to be optimized. The constraints imposed on the problem may be bounds on the state, a linear function (Ax), and a non-linear function ($c(x)$). These constraints are all bound by a lower-bound (l) and upper-bound (u) value; in the case of an equality constraint, l and u are simply equal to one another. The parameter x represents an n-dimensional vector of control variables that the NLP solver will tune. For example, a common spacecraft trajectory optimization NLP is to minimize the amount of propellant consumed onboard a spacecraft (the objective function), and a state for such an NLP may include the thruster's firing duration.

To solve an NLP, the parameters are tuned such that the objective function is minimized while adhering to all constraints. The solution is said to be "feasible" if the constraints are satisfied to within a user-defined tolerance $\delta_f \in \mathbb{R}^+$. The set of all feasible points is the feasible domain $D \subseteq \mathbb{R}^n$. Similarly, an "optimal" solution is one that has been proven to be a local minimum to within a tolerance of $\delta_o \in \mathbb{R}^+$. The goal of the NLP is to find a solution x^* that is both feasible and optimal.

It should also be noted that an NLP may sometimes aim to maximize instead of minimize the objective function. For maximization problem types, the NLP can be reformulated as a minimization problem by simply taking the negative of the objective function.

1.2 REAL-WORLD NLPS

NLPs are commonly found in spacecraft trajectory optimization problems and in other facets of aerospace engineering, but NLPs also extend into many other engineering research fields and business models. This section outlines a few examples of NLPs that occur in everyday, real-world situations. It is important to note that the problems listed have been drastically simplified for the sake of understanding the basics of NLPs in various fields; in reality, the objective functions are more complex and the number of constraints may easily reach into the hundreds in order to account for uncertainties and inefficiencies that are encountered in real-life scenarios.

1.2.1 Spacecraft Trajectory Optimization

As mentioned in Section 1.1: Non-Linear Program (NLP), one application of NLPs in aerospace engineering is determining the optimal trajectory of a spacecraft. A common objective function to be minimized is the propellant required for the spacecraft to reach its destination. The propellant mass can be calculated using Tsiolkovsky's rocket equation:

$$\Delta V = g_0 I_{sp} \ln \frac{m_i}{m_f} \tag{2}$$

where ΔV is the change in velocity required by the vehicle, g_0 is the gravitational acceleration due to gravity at Earth's surface (9.81 m/s^2), I_{sp} is the thruster's specific impulse, m_i is the spacecraft's initial mass, and m_f is the spacecraft's final mass after expending all propellant. By observing that the propellant mass is simply the difference between the spacecraft's initial and final mass, Equation (2) can be rearranged to solve for the objective function to be minimized:

$$f(x) = m_{prop} = m_i \left(1 - e^{\frac{-\Delta V}{g_0 I_{sp}}}\right) \tag{3}$$

where the variables to be optimized can be written (in a very simplified manner) as:

$$x = \begin{bmatrix} m_i \\ \Delta V \\ I_{sp} \end{bmatrix} \tag{4}$$

Two obvious constraints that may be imposed on this problem are the spacecraft's initial dry mass and the thruster's specific impulse; both constraints could be a set value or a range with a lower and upper bound. For space missions, finding an optimal solution that reduces the amount of propellant required onboard the spacecraft could save a substantial amount of money, especially

in terms of launch costs. Aerospace engineers often rely on NLP solvers to help plan feasible and cost-effective missions.

1.2.2 Product-Mix Problem with Price Elasticity

Another example of an NLP is the product-mix problem that commonly arises in business modeling. The relationship between demand and price of a product is an inverse curve $p(x)$; that is, the more demand there is for a product, the lower the price (Figure 1).

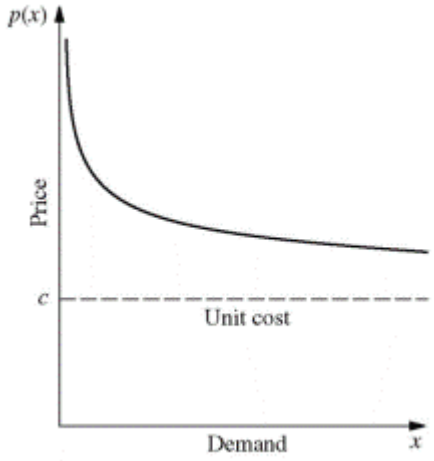


Figure 1. Price as a function of demand for the product-mix problem with price elasticity [2].

The profit for selling x units of the particular good is the sales revenue minus the production costs:

$$P(x) = xp(x) - cx \tag{5}$$

where $p(x)$ is the retail price of each unit and c is the manufacturing cost for each unit. Therefore, the company's total profit is the sum of each product's profits:

$$f(x) = \sum_{j=1}^{n} P_j(x_j) \tag{6}$$

The goal of the company is to maximize the total profits, and therefore $f(x)$ is the objective function of the problem. Since the profit for selling each good $P_j(x_j)$ is a non-linear function (as shown in Figure 1), the product-mix problem with price elasticity can be formulated as an NLP, and as such, the company's profits can be maximized using the definition outlined in Section 1.1: Non-Linear Program (NLP). Some constraints that may be imposed on this problem include an

upper limit to how many products can be manufactured in a given time and also a lower bound for the retail price (i.e. greater than $0) [2].

1.2.3 Portfolio Selection

For investors, NLPs can be beneficial in maximizing the return on investment. This problem is further complicated by the fact that risks are involved with any investment and therefore is not a trivial optimization problem. For example, an investor has $5,000 and plans to invest in two companies. Company 1 has an expected annual return of 20%, while company 2 has an expected annual return of only 16%. Let x_j denote the total investment in thousands of dollars for company 1 ($j = 1$) and company 2 ($j = 2$). The risk associated with both investments is given by:

$$risk = 2{x_1}^2 + {x_2}^2 + (x_1 + x_2)^2 \tag{7}$$

The expected return and risk can be combined into the objective function to be maximized:

$$f(x) = 20x_1 + 16x_2 - \theta[2{x_1}^2 + {x_2}^2 + (x_1 + x_2)^2] \tag{8}$$

where $\theta \geq 0$ is a constant associated with the impact of the risks on the total investment return. The objective function is subject to the following constraints [3]:

$$\begin{aligned} x_1 + x_2 &\leq 5 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned} \tag{9}$$

1.2.4 City Planning

A rather difficult optimization problem occurs when working as a city planner, especially when planning school districts. Consider a city that wants to build two high schools within the district. The town has a set of P subdivisions N_j. Let w_{1j} and w_{2j} represent the number of students from subdivision N_j attending schools A and B, respectively. The goal of this optimization problem is to minimize the distance travelled by all of the students assuming they may either attend school A or B. Therefore, the objective function can be written as follows:

$$f(x) = \sum_{j=1}^{P} w_{1j}\left((a - x_j)^2 + (b - y_j)^2\right)^{\frac{1}{2}} + w_{2j}\left((c - x_j)^2 + (d - y_j)^2\right)^{\frac{1}{2}} \tag{10}$$

where (a, b) and (c, d) are the coordinates of schools A and B, respectively, and (x_j, y_j) are the coordinates of the subdivision N_j modeled as a single point. Letting C_1 and C_2 denote the student capacity for schools A and B, respectively, and r_j denote the total number of students in each subdivision, the objective function is subject to the following constraints [3]:

$$\begin{aligned} \sum_j w_{1j} &= C_1 \\ \sum_j w_{2j} &= C_2 \\ w_{1j} + w_{2j} &= r_j \end{aligned} \tag{11}$$

1.3 CURRENT NLP SOLVERS

Several state-of-the-art NLP solvers currently exist and are used extensively by mission planners in the aerospace industry. One of the most popular is the Sparse Non-linear OPTimizer (SNOPT) developed by Gill et al [4]. Other NLP software packages include the Interior Point OPTimizer (IPOPT) [5] and "We Optimize Really Huge Problems" (WORHP). SNOPT, IPOPT, and WORHP employ Sequential Quadratic Programming (SQP), Interior Point (IP), and SQP-IP methods, respectively, for finding the solution of the NLP. Both of these approaches reformulate the NLP problem locally as a convex program and solve a sequence of these local programs to obtain a solution. SNOPT, IPOPT, and WORHP (among others) act as NLP solver plug-ins into spacecraft trajectory design tools, such as the Optimal Trajectories by Implicit Solutions (OTIS) developed at NASA Glenn and Boeing, the Evolutionary Mission Trajectory Generator (EMTG) and General Mission Analysis Tool (GMAT) currently under development at NASA GSFC, and the Mission Analysis Low-Thrust Optimization (MALTO) currently used at the Jet Propulsion Laboratory.

These existing solvers have been proven to solve large-scale NLPs; however, none of the solvers have employed parallel computing techniques, and therefore, depending on the size of the problem, may require hours, weeks, or even months to solve the NLP. The design tools mentioned, on the other hand, have started to implement parallel techniques to reduce the computational time required for such problems. As packages like OTIS, EMTG, GMAT, and MALTO continue to develop and become more efficient, the current serial NLP solvers will not be able to support a new parallel architecture and will frequently become an execution speed bottleneck. By creating a

parallelized NLP solver that is able to handle the robust architecture of the trajectory design tools, this bottleneck is substantially reduced and faster runtimes would benefit real-time collaborative mission design efforts such as the Mission Design Lab (MDL) process at the NASA Goddard Space Flight Research Center or the Team-X environment at the Jet Propulsion Laboratory, where fast generation of preliminary space mission trajectories is essential to the design process.

CHAPTER 2: NLPAROPT

Before developing the parallel code, a serial version of NLPAROPT (henceforth simply referred to as "NLPAROPT") was created as a baseline of comparison for testing against the other parallel NLPAROPT versions. The specific algorithms and methodology for an NLP solver were heavily researched and programmed into NLPAROPT. Several iterations of programming, debugging, refining, and testing took place over the course of several months in order to ensure that NLPAROPT was adequately efficient and, most importantly, accurate. A significant portion of the Phase I effort was devoted to the research and development of NLPAROPT since NLPAROPT_MPI and NLPAROPT_CUDA implement most of the original NLPAROPT code with parallel techniques scattered throughout. Therefore, it was imperative that NLPAROPT was close to completion (if not fully complete) before starting to program the parallel versions. This section outlines the NLPAROPT algorithm in detail, and subsequently, also provides an overall methodology behind the parallel versions. The specific parallel techniques for NLPAROPT_MPI and NLPAROPT_CUDA are outlined in CHAPTER 3: NLPAROPT_MPI and CHAPTER 4: NLPAROPT_CUDA, respectively.

2.1 ALGORITHM

The construction of algorithms for solving NLPs is based on satisfying the Karush-Kuhn-Tucker (KKT) conditions for convex programming. A convex program is a special type of NLP, whereby the objective function and constraints are formulated as convex functions. For a local optimization problem at a current parameter x with an objective function to be minimized $f(x)$, inequality constraints $g_i(x)$ $(i = 1, \dots, m)$, and equality constraints $h_j(x)$ $(j = 1, \dots, p)$, there exist a local minimum $\hat{x}$ and KKT multipliers $\lambda_i$and ν_j that satisfy the following conditions [6] [7] [8] [9]:

1. Stationary solution:

$$\nabla f(\hat{x}) + \sum_{i=1}^{m} \lambda_i g_i(\hat{x}) + \sum_{j=1}^{p} \nu_j h_j(\hat{x}) = 0 \tag{12}$$

2. Primal feasibility:

$$\begin{aligned} g_i(\hat{x}) &\leq 0 \; \forall \; i = 1, \dots, m \\ h_i(\hat{x}) &= 0 \; \forall \; j = 1, \dots, p \end{aligned} \tag{13}$$

3. Dual feasibility:

$$\lambda_i \geq 0 \; \forall \; i = 1, \dots, m \tag{14}$$

4. Complementary slackness:

$$\lambda_i g_i(\hat{x}) = 0 \; \forall \; i = 1, \dots, m \tag{15}$$

Once the local minimum $\hat{x}$ has been determined, this state can then be used as a guide for a new starting condition of another convex program to produce an improved solution. NLP algorithms gather the set of local solutions $\{\hat{x}_i\}$ that have been determined from the simpler convex problem and utilize yet another convex problem to converge to an improved local solution of the NLP, x^*. In other words, two logic levels must be implemented to solve an NLP: a *minor level*, which solves a local convex programming problem, and a *major level*, which provides a method for guiding the minor level process to find the local minimum of the real problem solution.

The algorithm implemented in NLPAROPT is a variation of an L_∞ exact penalty trust region method based off the work of Burke [10] and Yuan [11]. To utilize the penalty method, the constrained optimization problem (Equation (1)) must first be rewritten as a condensed, unconstrained optimization problem where all the inequality constraints are expressed in null-positive form:

$$NLP = \begin{cases} \min\limits_{x \in D \subseteq \mathbb{R}^n} f(x) \colon \mathbb{R}^n \to \mathbb{R} \\ subject\ to \quad c_i(x) = 0 \quad i = 1, \dots, m_e \\ \qquad\qquad c_i(x) \geq 0 \quad i = m_e + 1, \dots, m \end{cases} \tag{16}$$

where m_e is the number of equality constraints and m is the total number of constraints (both equality and inequality).

An L_∞ merit function can then be formulated as a convex combination of the objective function and constraints from Equation (16):

$$\Phi(x_k) = f(x_k) + \sum_{j=0}^{m_e} \mu_{kj} \max(0, c_j(x_k)) + \sum_{j=m_e+1}^{m} \mu_{kj} |c_j(x_k)| \tag{17}$$

where k is the number of major iterations and $\mu_{kj} \in \mathbb{R}^+$ is a penalty parameter determined by some user-defined logic at each major iteration. The last summation in Equation (17) considers constraints that are violated at the current parameter x_k, or "active" inequalities [12].

Yuan's trust region algorithm is an approximate nonsmooth minimization problem to determine the next guess of the solution $\hat{x}_k = x_k + s_k$, where s_k is the suggested step determined in the major iteration level (see Section 2.2: Major Level):

$$QP\ subproblem = \begin{cases} \min\limits_{x \in \mathbb{R}^n} \phi(x) = \nabla f_x^T s + \frac{1}{2} s^T B_k s + \sigma_k \|\tilde{c}_k + \nabla \tilde{c}_k^T s\|_\infty \\ subject\ to \quad \|s\|_\infty \leq \Delta_k \end{cases} \tag{18}$$

where $\tilde{c}_k$ is an active constraint, σ_k is the penalty parameter, Δ_k is the trust region radius, and B_k is the approximation for the Hessian of the Lagrangian [11] [12]. Trust region methods, such as the one proposed by Yuan, are generally more reliable than line search algorithms and allow both the direction and length of the step to vary [11] [13]. Figure 3, Figure 2, and Figure 4 illustrate the start-up, major level, and minor level algorithms, respectively. The details of each of these levels will be discussed in the following sections.

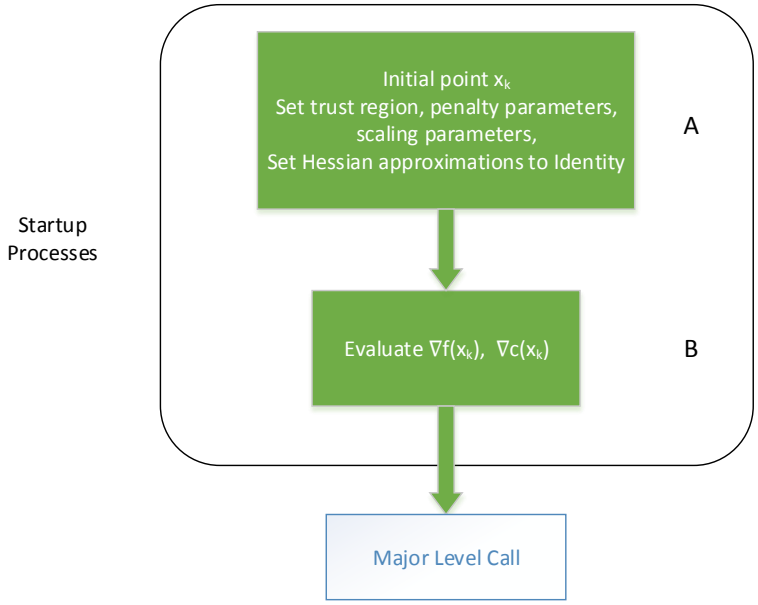


Figure 2. Start-up algorithm [14].

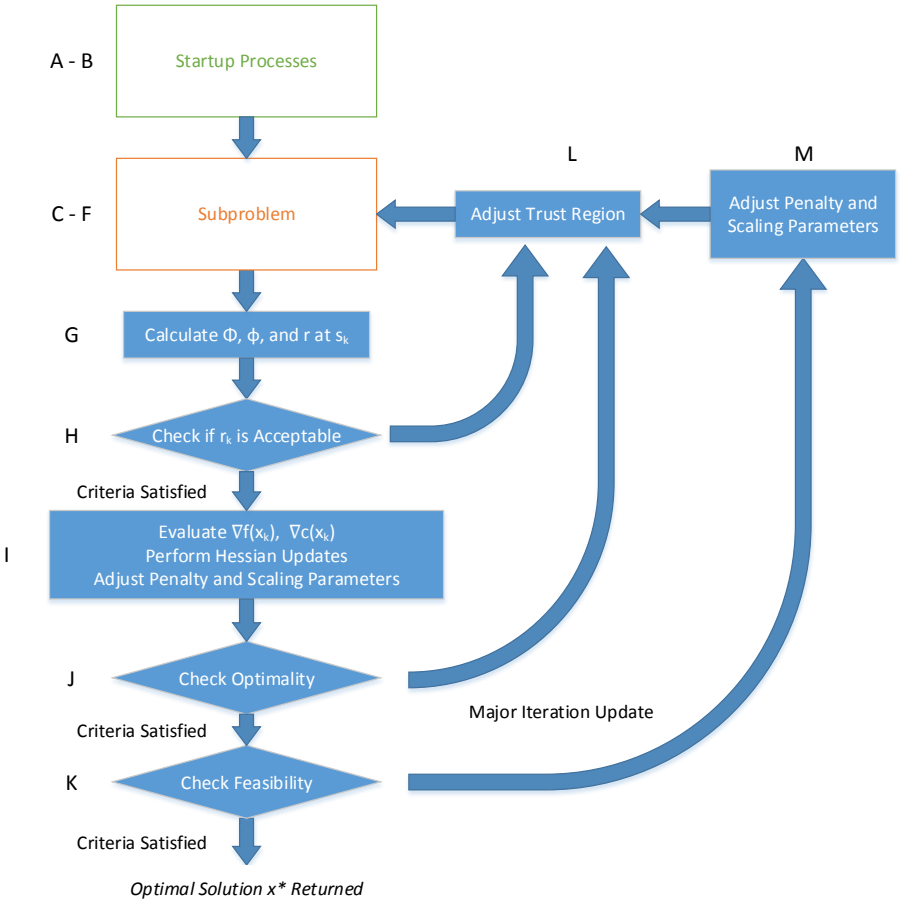


Figure 3. Major level algorithm [14].

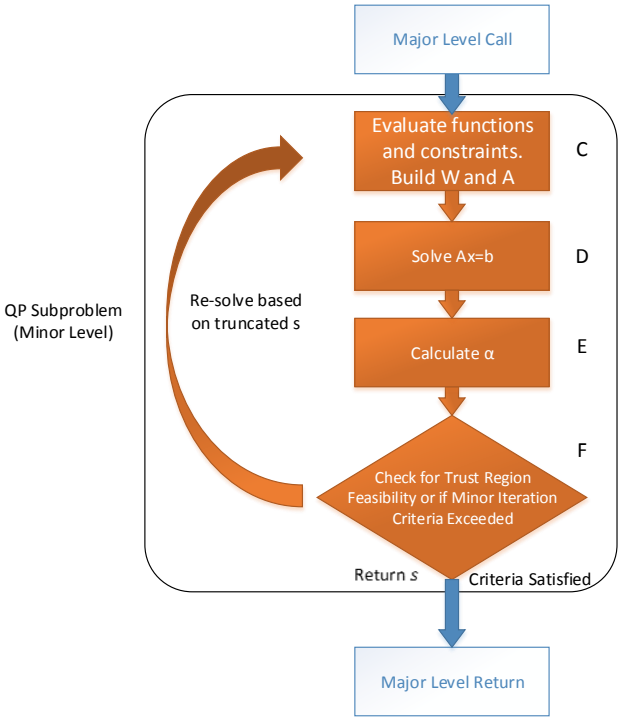


Figure 4. Minor level algorithm [14].

2.2 MAJOR LEVEL

As previously mentioned in Section 2.1: Algorithm, the purpose of the major level is to determine if the suggested step s_k from the minor level is acceptable and form a new local subproblem, or if it should be rejected and therefore corrected. The step quality factor r_k quantitatively determines whether the step is acceptable or not:

$$r_k = \frac{\Phi(x_k) - \Phi(\hat{x}_k)}{\phi(0) - \phi(s_k)} \tag{19}$$

where x_k is the current state and $\hat{x}_k = x_k + s_k$ is the suggested new state. The numerator and denominator in Equation (19) represent the actual change and the predicted change in the objective function, respectively. If the step quality factor is near unity, the approximation is deemed accurate enough, and therefore the trust region is increased to allow more points to be considered for the next step. However, the approximation is not considered accurate if the step quality factor is small or negative, and the new trust region Δ_{k+1} is reduced according to the following parameters [12]:

$$\Delta_{k+1} = \begin{cases} max\{2\Delta_k, 4\|s_k\|_\infty\} & 0.9 < r_k \\ \Delta_k & 0.1 \le r_k \le 0.9 \\ max\left\{\frac{\Delta_k}{4}, \frac{\|s_k\|_\infty}{2}\right\} & r_k < 0.1 \end{cases} \tag{20}$$

2.3 MINOR LEVEL

Before entering the minor level General QP (GQP) subproblem solver, Equation (18) must first be slightly reformulated:

$$GQP = \begin{cases} \min\limits_{s \in \mathbb{R}^n} f_x + \nabla f_x^T s + \frac{1}{2} s^T B_k s \\ subject\ to \quad l \le \begin{pmatrix} s \\ As \end{pmatrix} \le u \end{cases} \tag{21}$$

An active set algorithm was implemented as the QP subproblem solver, and the central part of the solver is the linear system solve (represented as $Ax = b$ in block D of Figure 4), which has the following Lagrange-Newton form:

$$Ax = \begin{bmatrix} B_i & W^T \\ W & 0 \end{bmatrix} \begin{bmatrix} s_k \\ \lambda_k \end{bmatrix} = \begin{bmatrix} \nabla f_k \\ \tilde{c}_k \end{bmatrix} = b \tag{22}$$

where matrix A is composed of the Hessian approximation B and working set W, which is the gradient of the active constraints at the current point x_k. The subscript k represents the minor iteration count and λ_k are the dual variables associated with the active constraint [12].

The goal of each minor level iteration is to solve for $x = \begin{bmatrix} s_k \\ \lambda_k \end{bmatrix}$, which then provides a suggested step and dual variable to satisfy the active constraints and minimize the objective function. A line search using the gradient method is then performed to determine a new viable point. The search will terminate if the step size to reach the new point is less than a defined tolerance ε; at this point, the algorithm determines that because the search is hardly straying from the previous point, the current point must be close enough to the optimal point. The gradient method is outlined in the following procedure:

1. For $k = 0$: Select x_0, ε
2. For $k > 0$:
    a. Compute $\alpha_k = arg\,min_\alpha\, f(x_k + \alpha s_k)$, where $\alpha \in [0,1]$
    b. Set $x_{k+1} = x_k + \alpha_k s_k$
3. For $k = k + 1$: Stop if $\|\alpha_{k+1} s_{k+1}\| < \varepsilon$, otherwise repeat step 2

Any violated constraints from the line search are added to W, and similarly, any constraints that are no longer violated are removed. The line search continues until the tolerance conditions are met or the maximum number of iterations has been achieved [12].

2.4 HESSIAN UPDATE

For both the minor and major level, a Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula [15] was implemented to perform a symmetric rank-2 (SR2) update to compute the approximate Hessian matrix B_k. An SR2 update was used to avoid the high cost for computing the Hessian matrix $\nabla \cdot \nabla f(x)$, especially for large problems that require many Hessian entries. Though the approximation results in superlinear convergence, successive updates provide a subproblem that converges quadratically; therefore, it can be verified that B_k does approach a single value, specifically $F'(x^*)$ [16]. The SR2 update to the approximate Hessian B_k is defined as follows:

$$B_{k+1} = B_k + \frac{y_k {y_k}^T}{{y_k}^T s_k} - \frac{B_k s_k {s_k}^T B_k}{{s_k}^T B_k s_k} \qquad (23)$$
$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$

If the $y_k^T s_k$ term is very small, numerical issues can arise that result in an inaccurate Hessian update. To avoid this problem, an additional check is imposed to determine if this term is too small and, if so, the second term in the SR2 update equation $\left(\frac{y_k {y_k}^T}{{y_k}^T s_k}\right)$ is dropped. Otherwise, if B_k is symmetric positive definite and $y_k^T s_k \geq 0$, the Hessian update should be stable, without singularities, and always produce symmetric positive definite matrices.

2.5 LAPACK/BLAS ROUTINES

The Linear Algebra PACKage (LAPACK) is a set of routines that perform various linear algebra calculations, including solving systems of linear equations, least-squares solutions of linear systems of equations, and eigenvalue problems [17]. These routines are executed by calling on the Basic Linear Algebra Subprogram (BLAS) functions, which are a series of subroutines that are divided into three levels depending on the complexity of the scalar, vector, or matrix calculation to be performed: Level 1 contains vector operations, Level 2 contains matrix-vector operations, and Level 3 contains matrix-matrix operations [18].

NLPAROPT implemented several Level 1 and Level 2 BLAS routines in order to increase the overall efficiency of the code when performing linear algebra calculations; these routines are listed in Table 1. By using BLAS routines, computationally expensive double-, triple-, and even quadruple-nested for-loops were completely eliminated and replaced with efficient, robust calls to the BLAS routines. The Hessian update (see Section 2.4: Hessian Update) benefited greatly from such routines, as the BFGS SR2 update algorithm for computing the approximate Hessian is purely a series of vector-vector and matrix-vector multiplications. All BLAS routines used in NLPAROPT are of a double type, as indicated by the *d* at the beginning of each routine name.

Table 1. NLPAROPT BLAS routines [18].

Level	Routine	Description
1	ddot	Computes the dot product of two vectors
2	dspmv	Symmetric packed matrix vector multiply
2	dspr	Symmetric packed rank 1 operation: $A = \propto xx'$

CHAPTER 3: NLPAROPT_MPI

As mentioned in CHAPTER 2: NLPAROPT, NLPAROPT was first fully developed and tested before programming the MPI-based parallel version, NLPAROPT_MPI. This order was chosen due to the fact that NLPAROPT_MPI inherits most of the algorithm and methodology behind NLPAROPT; the only difference in the code is Parallel BLAS (PBLAS) routines replaced the original BLAS routines, and some parallel operations were added to replace blatantly serial operations that could significantly benefit from computational speedups that parallelization provides. For example, one operation in NLPAROPT is determining the worst violated constraint, i.e. the constraint whose value at the current point deviates the most from the desired value. In NLPAROPT, the processor would cycle through each constraint and then determine which has the highest value, whereas in NLPAROPT_MPI, the constraints can be parsed among all the processors; each processor determines its local worst violated constraint, and then one designated processor compares all of the local worst violated constraints to determine the global worst violated constraint. The following sections outline the implementation of parallel techniques in NLPAROPT_MPI utilizing Boost.MPI, a library used for processor communication, and PBLAS.

3.1 BOOST.MPI

The library chosen for parallelization is Boost.MPI, which allows for one or more processes to communicate by either sending or receiving messages (point-to-point communication) or by coordinating as a group (collective communication). Unlike the standard C- and Fortran77-based Message Passing Interface (MPI) library, Boost.MPI is C++-friendly and supports user-defined data types, C++ Standard Library types, arbitrary function objects for collective algorithms, as well as the use of modern C++ library techniques to maintain maximal efficiency [19]. The Boost.MPI collective operations used in NLPAROPT_MPI are outlined in Table 2.

Table 2. NLPAROPT_MPI Boost.MPI operations [19].

Operation	Description
reduce	Summarizes the values from each process into a single value by one designated processor
all_reduce	Performs the “reduce” operation and broadcasts the single value to all processors
all_gather	Collects values from every process into a vector at one designated processor
broadcast	Broadcasts a value from a single process to all other processors

3.2 ScaLAPACK/PBLAS Routines

In addition to the Boost.MPI library, parallelization was also implemented in NLPAROPT_MPI through the use of PBLAS. PBLAS routines are capable of performing the same linear algebra operations as BLAS but, as the name suggests, are intended for parallel distributed memory architectures. As the Linear Algebra Package (LAPACK) software library relies on BLAS routines, the Scalable LAPACK (ScaLAPACK) software library relies on PBLAS routines.

ScaLAPACK is a dense linear system solver and uses a two-dimensional block-cyclic data distribution scheme. Two parameters, the row blocking factor (MB) and the column blocking factor (NB), define how a data set is distributed over a number of processors. The blocking factor determines how many rows or columns are assigned to each block, while the processor grid determines how often new blocks are assigned to processors. An example is shown in Figure 5, where a 9x9 global matrix is distributed over six processors in a 2x3 processor grid with a square blocking factor of two. The example shown illustrates how the data is distributed across all processors; Processor 0 (red) has a local matrix of 20 elements, whereas processor 5 (white) has 8 elements. Though it may seem as though the distribution is incredibly uneven, the two-dimensional block-cyclic distribution consistently provides the most even distribution relative to other schemes, such as the one-dimensional block-cyclic distribution scheme [20].

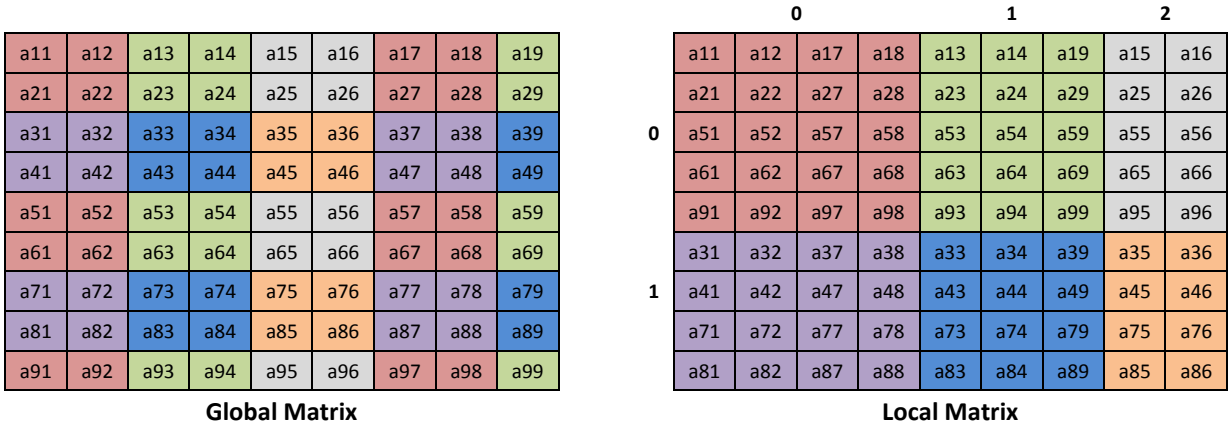


Figure 5. Two-dimensional block-cyclic data distribution of a 9x9 global matrix over a 2x3 processor grid using square blocking factor of 2.

Several PBLAS routines belonging to the first two levels were used in NLPAROPT_MPI to perform various linear algebra routines and are outlined in Table 3. Each routine used in NLPAROPT_MPI is of a double precision data type, indicated by *d* as the second character, and the Level 2 routines have a general matrix argument type, indicated by *ge* [21]. Though the Level

2 routines can be implemented in a symmetric matrix format instead of general (i.e. packed), which subsequently would decrease the amount of elements stored for each linear algebra calculation, the Phase I NLPAROPT team ran into issues that occurred while attempting to communicate among processors using MPI with only portions of a matrix. Specifically, when parsing out elements of a matrix to multiple processors, the communication process requires all elements of the matrix to be stored. For a parallel program such as NLPAROPT_MPI, it was determined that using a general matrix format is currently the only option.

Table 3. NLPAROPT_MPI PBLAS routines [21].

Level	Routine	Description
1	pddot	Calculates the dot product of two distributed real vectors
2	pdgemv	Matrix-vector product
2	pdger	Rank-1 update of a matrix
2	pdgesv	Computes solution to a linear equation Ax=b

CHAPTER 4: NLPAROPT_CUDA

Though Graphics Processing Units (GPUs) have traditionally been used for the purpose of creating life-like computer-generated graphics, NVIDIA has recently developed its own GPU-based parallel computing platform (CUDA) that takes advantage of the thousands of GPU cores, rather than relying only on four to eight cores that standard consumer Central Processing Units (CPUs) contain [22]. Though the NLPAROPT research team had minimal experience in GPU-based programming prior to working on NLPAROPT, the team was intrigued by the prospect of learning a relatively new programming technique and running a CUDA variant of NLPAROPT against NLPAROPT_MPI to determine if there was a significant change in performance between the two parallelization methods. Similar to NLPAROPT_MPI, the second parallelized version of NLPAROPT, NLPAROPT_CUDA, was developed after the serial code and algorithm were complete and certain sections of the serial code were replaced with CUDA functions, specifically to perform linear algebra calculations provided by cuBLAS and to simplify and speed up the Hessian update.

4.1 CUDA

CUDA is a GPU-based parallel programming platform that allows the programmer to write C programs that interface with the GPU using CUDA extensions. Due to time constraints, CUDA techniques were only implemented in the Hessian update (see Section 2.4: Hessian Update); however, it can be argued that this is one of the most computationally-intensive steps in the NLPAROPT code and therefore should show a noticeable speed-up when tested against the serial NLPAROPT. A list of the CUDA functions the team programmed into NLPAROPT_CUDA is outlined in Table 4.

Table 4. NLPAROPT_CUDA CUDA functions [21].

Routine	Description
hessianCUDAIdentity	Produces the identity matrix in an nxn array
calcY	Performs the $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ calculation in the Hessian update
zeros	Zeros out all elements of the array
addResults	Adds three matrices together
copyHtoB	Copies the H sub-matrix of the Hessian array to CPU memory
copyBtoH	Copies the GPU copy of the H sub-matrix back into the Hessian array on the CPU

4.2 cuBLAS Routines

The linear algebra library deployed on the GPU is cuBLAS, the CUDA-based version of BLAS. One striking benefit of cuBLAS is its ability to perform linear algebra calculations on symmetric matrices using only an upper- or lower-triangular matrix form; for example, in the linear system solve (see Section 2.3: Minor Level), the matrix A is stored in an upper-triangular form, and therefore, for a given matrix dimension n, only $n^2 - \sum_{i=1}^{n} i$ elements must be stored in the matrix instead of n^2 for a packed matrix like PBLAS requires. Similar to BLAS, the cuBLAS routines are of the double precision data type (indicated by a *D* in the routine name). The cuBLAS routines implemented in NLPAROPT_CUDA are outlined in Table 5.

Table 5. NLPAROPT_CUDA cuBLAS routines [23].

Level	Routine	Description
1	cublasDdot	Calculates the dot product of two distributed real vectors
2	cublasDsymv	Matrix-vector product for a symmetric matrix
2	cublasDsyr	Rank-1 update of a symmetric matrix

CHAPTER 5: NLPAROPT_MPI EFFICIENCY ANALYSIS

As with any program, parallel techniques designed to improve the efficiency of the program can potentially lead to significant runtime inefficiencies. Inefficiencies, or bottlenecks, often arise when the work is not divided evenly among all processors, and as a consequence, a lightly-loaded processor (or processors) must wait for other heavily-loaded processors to "catch up" before continuing on to the next operation. This increased idle time is highly inefficient, and ultimately defeats the purpose of utilizing parallel programming techniques. Another inefficiency in parallel programming arises when the program is run on too many processors, thereby increasing the processor-processor communication time and also the overall computational runtime. Once the prototype NLPAROPT_MPI was complete, it was important to determine if such inefficiencies existed in the code, and if so, develop a method to reduce or completely eliminate these inefficiencies. In order to analyze the parallel features of NLPAROPT_MPI, such as determining when processors are communicating with one another, when processors are waiting for others to "catch up", etc., a logging library called MPI Parallel Environment (MPE) was implemented in certain areas of the NLPAROPT_MPI code. MPE outputs a logging file after the code is run, which can then be read by the visualization software, Jumpshot-4, to display the MPI calls for each processor as a function of time. From this visualization software, in conjunction with several algorithms developed in MATLAB, an analysis of the efficiency of NLPAROPT_MPI was conducted.

It is important to note that an efficiency analysis was not conducted on NLPAROPT_CUDA. After the completion of the SBIR Phase I contract in December 2014, the NLPAROPT team decided that all further development should be focused solely on NLPAROPT_MPI. If CUDA techniques are implemented back into NLPAROPT in the future, the methods for conducting an efficiency analysis outlined in this section are still applicable for NLPAROPT_CUDA.

5.1 MPI PARALLEL ENVIRONMENT (MPE)

MPE is a library of extensions used with MPI programs that include several profiling libraries for post-mortem visualization of parallel programs. When the program is linked with the MPE logging library, all MPI calls are intercepted and a log file is generated that includes the timestamp and event type for each MPI call in a Common LOG (CLOG) format, which stores data

in a binary format compared to the original ASCII LOG (ALOG). In the default setting, all MPI calls for each processor are stored in the output log file, though this has proven to generate incredibly large log files on the order of gigabytes when implemented with NLPAROPT_MPI. An alternative is manually implementing MPE logging calls throughout the program such that only certain parallel events of interest are stored in the log file. Once the log file has been generated, the user can then import the log file into a visualization software tool (in this case, Jumpshot-4) to understand and analyze how and when the processors are communicating with one another and where improvements can be made [24].

5.2 JUMPSHOT-4

Note: This section serves as an introduction to using the Jumpshot-4 parallel visualization software. However, it only touches on a fraction of the programs and settings that were specifically used for NLPAROPT_MPI. For more information on Jumpshot-4's full capabilities, see the Jumpshot-4 Users Guide [25].

After the CLOG log file has been generated by MPE, Jumpshot-4 is then used to first convert the CLOG file to a Scalable LOG file (SLOG), which is the ideal format chosen to help visualization programs handle very large log files. The SLOG format stores the CLOG events as states (essentially an event with a duration). After this conversion, the SLOG file is imported into Jumpshot-4 and a diagram of colorful states and communication arrows is generated.

Figure 6 shows the diagram produced with NLPAROPT_MPI run on two processors using the default setting to log all MPI calls. A color-coded legend is on the left side of the screen, and the illustration on the right side of the screen is a plot of the different state of each processor for a certain time period. Each rectangle represents a state, and the colored blocks inside each rectangle represent specific events. The top and bottom rows of states correspond to Processor 0 and Processor 1, respectively, as indicated by the y-axis labels on the left side of the diagram. The x-axis of the diagram is the time in seconds. When the Jumpshot-4 visualization is launched, the timeline spans the entire duration of the program (in the example in Figure 6, the run duration is 460 seconds). Jumpshot-4 allows the user to zoom-in to the desired events to within microseconds, as shown in the bottom two screenshots in Figure 6 [25].

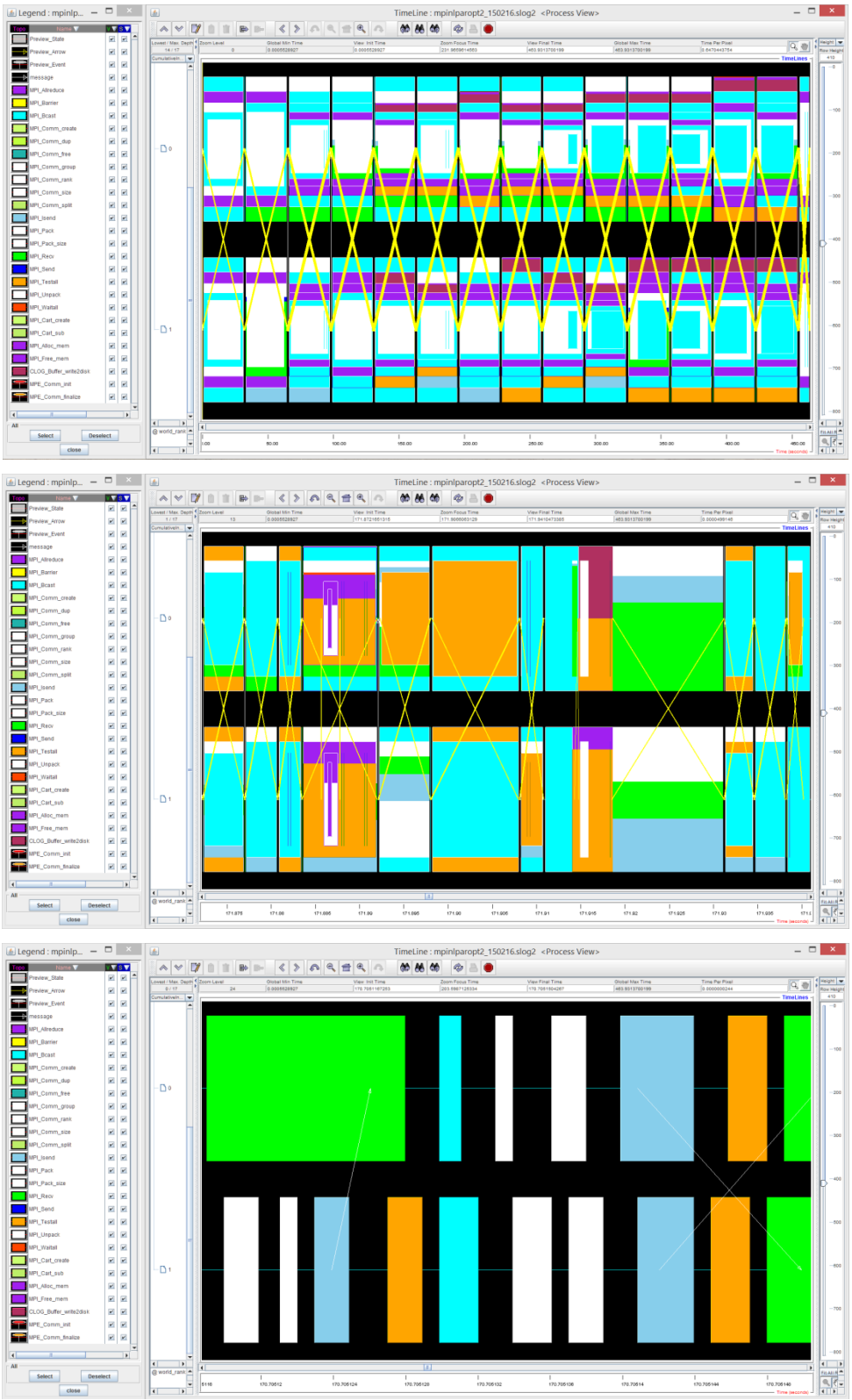


Figure 6. Jumpshot-4 visualization of NLPAROPT_MPI run with two processors zoomed out completely to a 460-second window (top), zoomed in to a 0.07-second window (middle), and zoomed in to a 0.00003-second window (bottom).

The different colors of the events inside each state indicate a different MPI call. The legend on the left side of the screens in Figure 6 show each call, such as "send", "receive", and "broadcast", and its associated color. For example, a common color in Figure 6 is turquoise, which corresponds to a broadcast command. There are several different preview settings to represent MPI calls for a given state duration. The setting chosen in Figure 6 is called "Cumulative Inclusion Ratio", which nests all events inclusively. An example state is shown in Figure 7. With this setting type, the events are arranged such that the event that contributes the most time to the total duration of the state is the tallest strip. In Figure 7, there are two strips that are the tallest (the green "receive" event and the white "pack size" event) and therefore consume approximately the same amount of time for the duration of the state. The event(s) with shorter durations are nested inside the larger strips and their size is determined based on the ratio of time the event contributes to the state compared to the largest events. For example, in Figure 7, the next largest event is the purple "all reduce" event. Considering the fact that all nested states include the area within the nested events, the purple event accounts for approximately 80% of the state's total area, and therefore contributes 80% of the time the largest green and white events contribute (which equate to 100% time). This inclusion ratio scheme causes some confusion; since the most time-intensive events are considered to have a ratio of 1 (or 100%) and the nested events are considered to have a ratio < 1 depending on the time contribution compared to the largest events, the total time ratio of the state is guaranteed to be ≥ 1 [25].

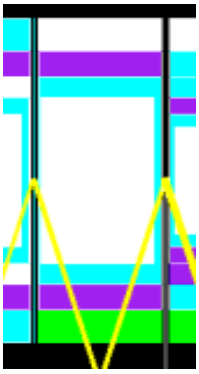


Figure 7. One state with the "Cumulative Inclusion Ratio" preview setting.

The yellow arrows (seen as lines due to the overlapping of hundreds of arrowheads in Figure 6) indicate how many communication calls occur between each processor for the given time frame. The thickness of the arrows indicates how many communications have occurred; in the example in Figure 6, the PREVIEW_ARROW_LOG_BASE setting is set to 10. For example, the preview arrows in the middle states are about twice as thick as those in the first state (on the far left), and thus there are about ten times more processor-processor communications occurring in the middle states than in the first state [25].

The purpose of creating a visualization of NLPAROPT_MPI is to determine if there are any inefficiencies that have been created with the implemented parallel techniques in order to hone

in and fix these areas. However, by observing Figure 6, which plots all of the MPI calls in the entire program, it is nearly impossible to understand what exactly is going on between each processor, let alone determine where any potential inefficiencies may occur, even when zoomed in as much as possible. Though the default setting for MPE is to log all MPI calls, it is also possible to manually implement MPE logging functions throughout the code to target only certain MPI calls, thereby producing a much cleaner visualization that is easier to comprehend [24].

MPI calls occur both in the major and minor level of NLPAROPT_MPI. By logging the MPI calls in various parts of both levels, the Jumpshot-4 visualization would show the time stamp and the durations for each of these functions for all processors and make it possible to see where certain processors may be completing the MPI functions quicker than others and therefore idling before exiting the loop and entering the next loop. Three functions were chosen to be logged by MPE for this analysis: the Hessian update in the major level, the construction of the A matrix in the minor level, and checking for the trust region feasibility using the "find worst violated constraint" function in the minor level (shown in Figure 8 and Figure 9, adapted from Figure 3 and Figure 4, respectively). Though there are several functions within the major and minor level to log, these three were chosen somewhat arbitrarily; as long as the timing of each of these functions lines up with each processor and there is little to no wait time in between the outer and inner loops, these three functions should indicate that the efficiency of NLPAROPT_MPI as a whole is sufficient. Conversely, if the Jumpshot-4 visualizations of these three events show the duration of each function for each processor greatly differs, NLPAROPT_MPI is not entirely efficient, as some processors may be idling while waiting for other processors to complete the function.

The Jumpshot-4 visualizations of the major level Hessian update, the minor level construction of the A matrix, and the minor level "find worst violated constraint" function are shown in Figure 10, Figure 11, and Figure 12, respectively. These functions were initially only logged for a two-processor NLPAROPT_MPI run to make the visualization as comprehensible as possible. To analyze the efficiency of NLPAROPT_MPI, the visualizations for each function were zoomed in to only display one event (the bottom of the three screenshots in each figure). By zooming in to view only one event, it can be determined exactly how much time each processor takes to complete the function. As shown in the zoomed in figures, it is evident that one processor requires more time than the other; however, this difference is on the order of milliseconds, and in some cases, microseconds. Because these differences are so small, it can be concluded that

NLPAROPT_MPI is efficient in the sense that processors are waiting an almost negligible amount of time for others to complete the current function, at least for the two-processor runs. However, NLPAROPT_MPI was developed to be run on more than just two processors, and depending on the size of the non-linear problem to be optimized, can potentially run on hundreds or thousands of processors. Therefore, though NLPAROPT_MPI seems sufficiently efficient when run on just two processors, it is important to analyze the efficiency using more processors.

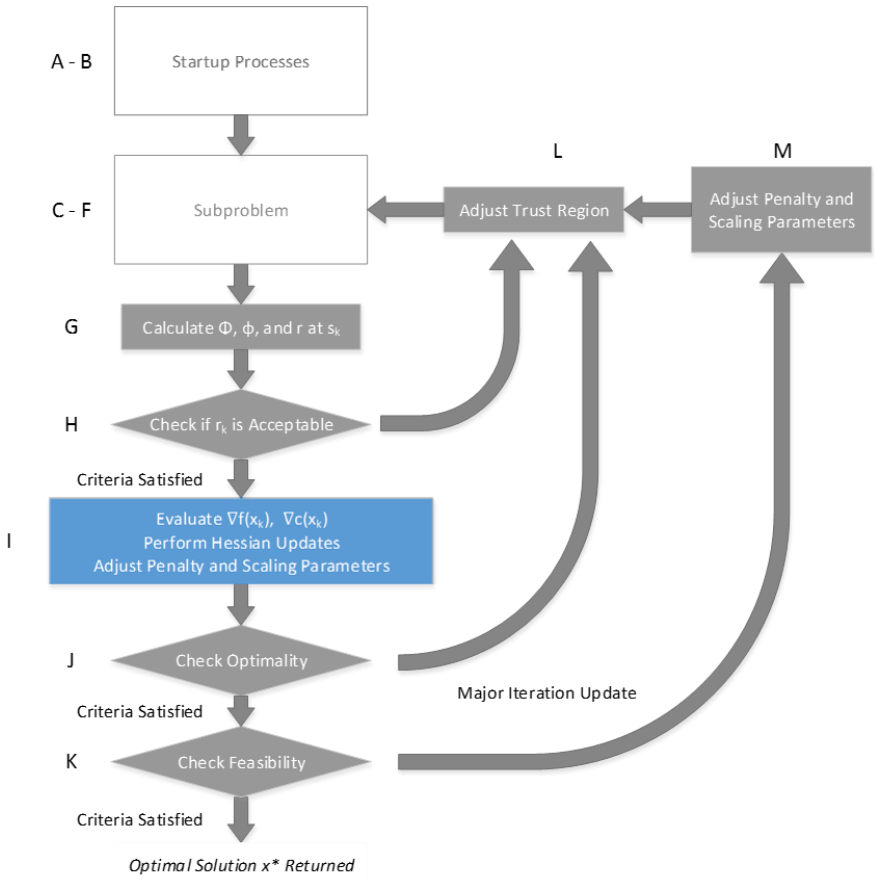


Figure 8. Major level algorithm with areas highlighted to be visualized in Jumpshot-4 [14].

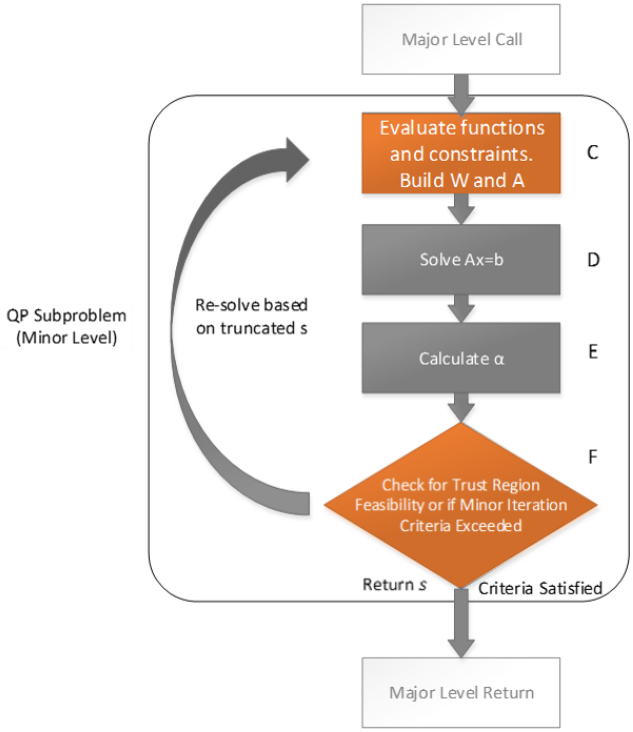


Figure 9. Minor level algorithm with areas highlighted to be visualized in Jumpshot-4 [14].

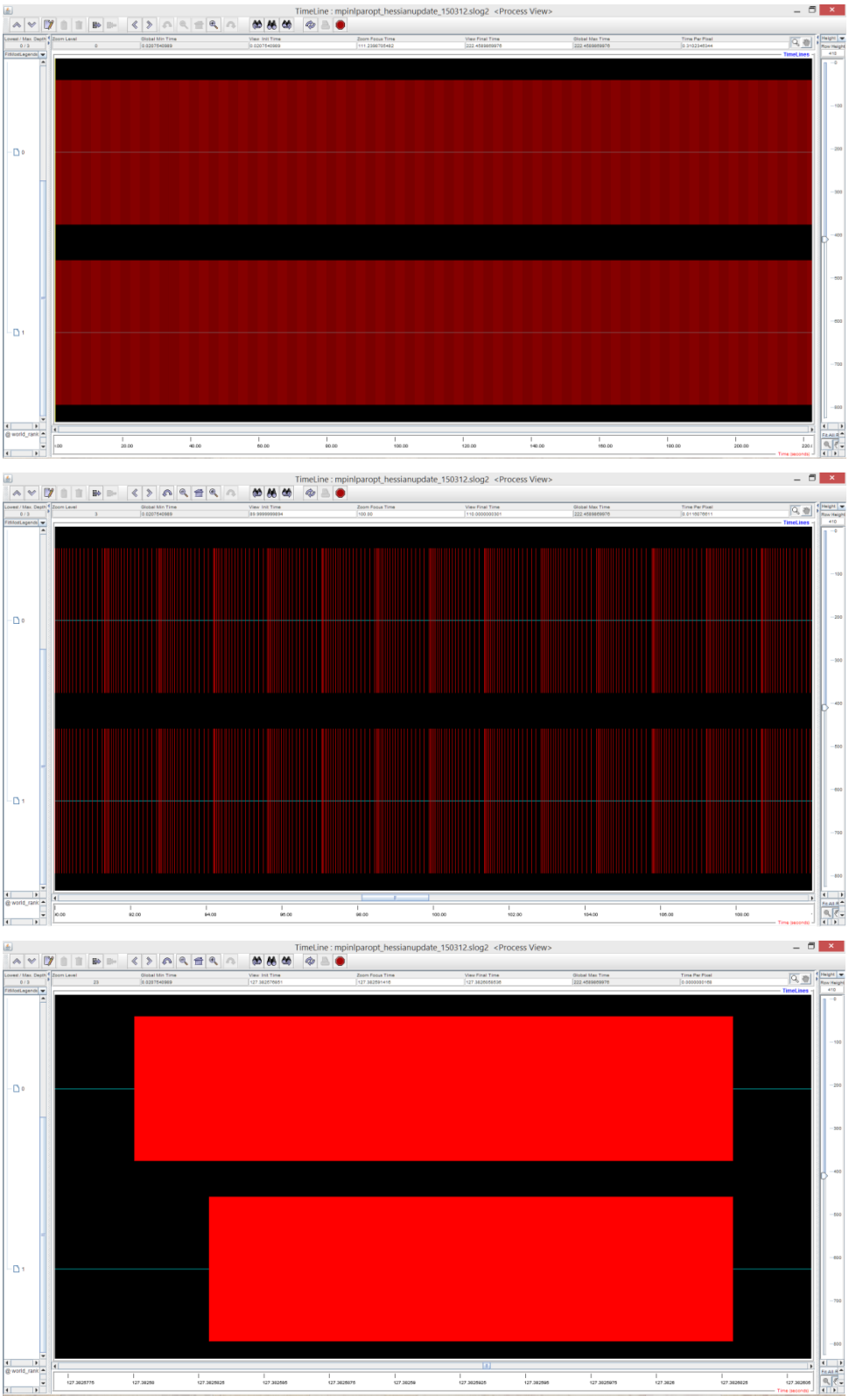


Figure 10. Jumpshot-4 visualization of NLPAROPT_MPI run with two processors manually logging the major level Hessian update, zoomed out completely to a 226-second window (top), zoomed in to a 20-second window (middle), and zoomed in to show one event (bottom).

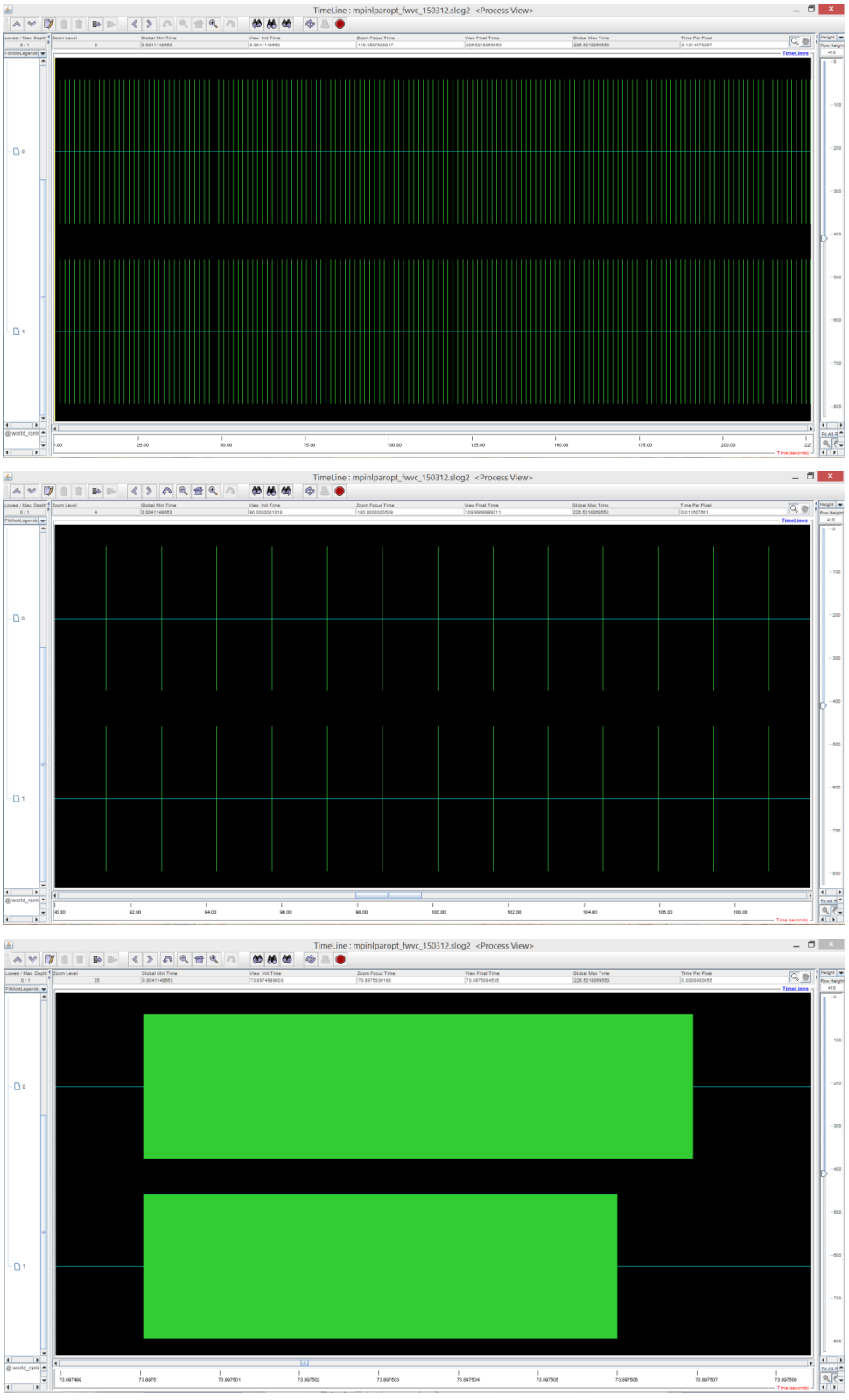


Figure 11. Jumpshot-4 visualization of NLPAROPT_MPI run with two processors manually logging the minor level "find worst violated constraint" function, zoomed out completely to a 226-second window (top), zoomed in to a 20-second window (middle), and zoomed in to show one event (bottom).

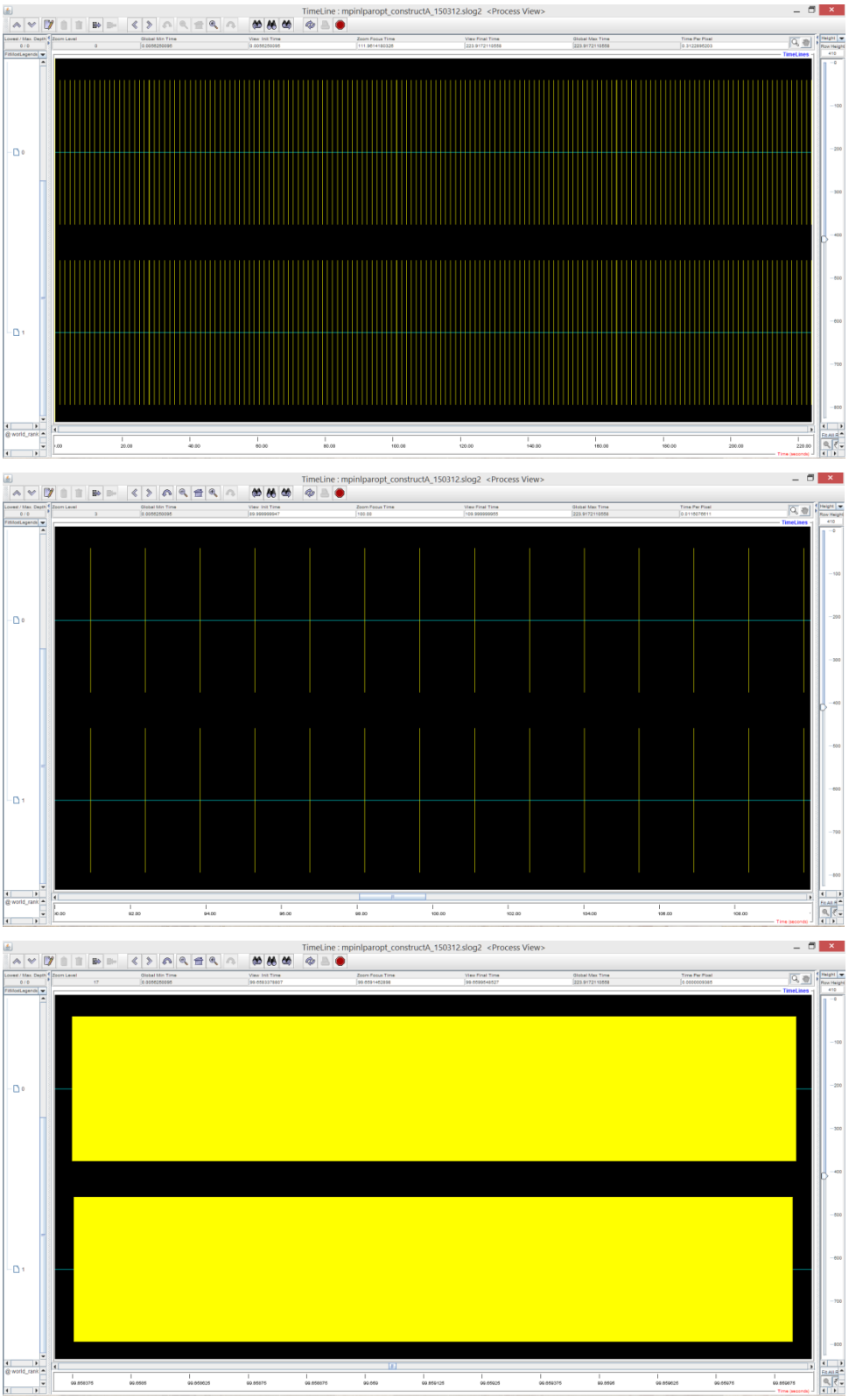


Figure 12. Jumpshot-4 visualization of NLPAROPT_MPI run with two processors manually logging the minor level construction of the A matrix, zoomed out completely to a 226-second window (top), zoomed in to a 20-second window (middle), and zoomed in to show one event (bottom).

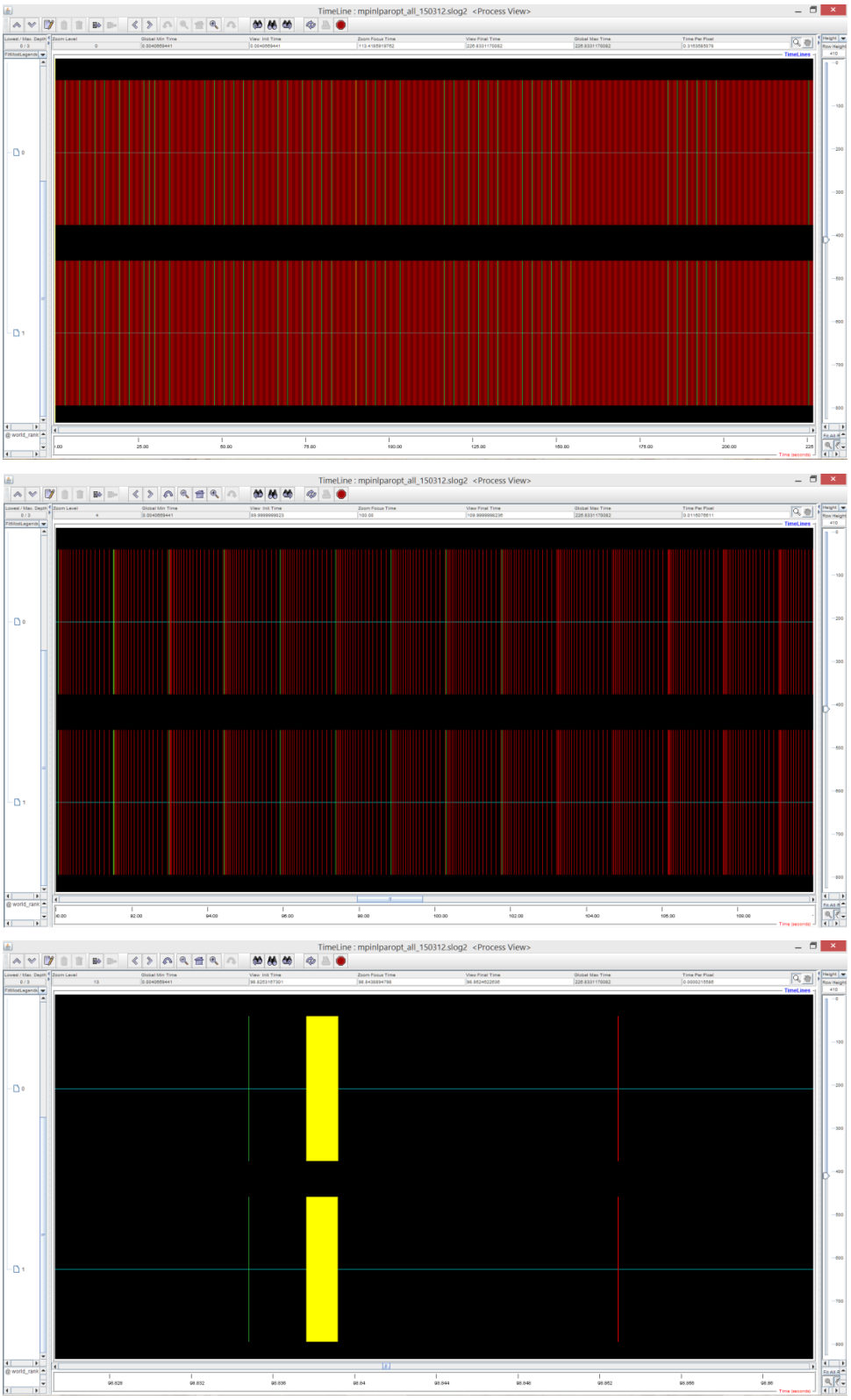


Figure 13. Jumpshot-4 visualization of NLPAROPT_MPI run with two processors manually logging all three major and minor level functions, zoomed out completely to a 226-second window (top), zoomed in to a 20-second window (middle), and zoomed in to show one series of events (bottom).

5.3 TEST PROBLEM

The capstone problem used to run test cases for the efficiency analysis was the Multiple Shooting Test Problem (MSTP). The MSTP was inspired by a classical optimal control theory problem that can be found in Bryson and Ho [26]. The goal of the MSTP is to transfer a spacecraft from an initial circular orbit to a final co-planar circular orbit of higher altitude using continuous thrust in fixed-time.

A multiple forward shooting transcription was chosen to parameterize the trajectory into N segments and cast it as a non-linear program. At the start of each segment, the optimizer selects values for the spacecraft's state vector $X = [x, y, \dot{x}, \dot{y}, m]^T$ and thrust pointing angle γ, where (x, y) is the spacecraft's position, $(\dot{x}, \dot{y})$ is the spacecraft's velocity, and m is the spacecraft's mass. Equality constraints are imposed on the states ending one segment and the initial states of the next segment to ensure continuity between each step. Two additional equality constraints are imposed to ensure the initial and final orbits are circular. The objective function is to maximize the final orbit radius. A depiction of the MSTP discretization is shown in Figure 14. The arrows represent the magnitude and direction of thrust.

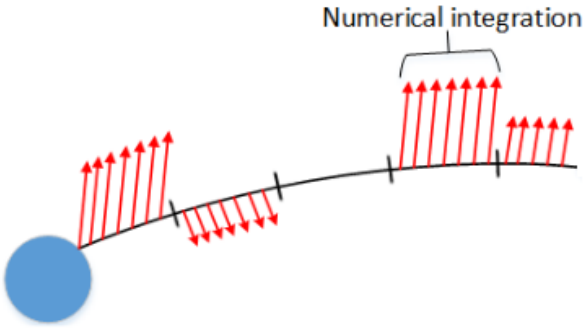


Figure 14. Representation of a forward-integrated trajectory discretized using multiple shooting.

This test problem was chosen due to its real-world application in aerospace mission planning as well as scalability. Though the MSTP is capable of optimizing a complex trajectory with effectively infinite segments, the user can also choose a more reasonable number of parameterized segments to decrease the total runtime while still optimizing a relatively complex non-linear program.

5.4 TEST HARDWARE

All test cases for the efficiency analysis were conducted on the campus-wide Taub cluster at the University of Illinois at Urbana-Champaign (UIUC) (Figure 15). Taub features 512 nodes with 12 Intel HP X5650 2.66 GHz 6C processors each, which provided a robust and scalable parallel computing environment for preliminary tests of NLPAROPT_MPI [27].

Figure 15. Campus cluster [27].

5.5 EFFICIENCY ANALYSIS RESULTS

In any parallel program, two types of inefficiencies exist: an inefficiency due to the increased communication time that is required as the number of processors increases, and an inefficiency in increased processor idling time due to an uneven distribution of the workload among the processors. The former is unfortunately an inevitable consequence of parallel programming, though it is important to determine how this communication inefficiency affects the program's runtime and develop a method for choosing the optimal number of processors that minimize the impact of this inefficiency. The latter inefficiency is dependent on how the programmer has implemented the parallel distribution scheme and what blocking factors are being used; in the case of NLPAROPT_MPI, a two-dimensional block-cyclic distribution scheme is utilized, and though this scheme consistently distributes elements of a matrix more evenly than other distribution schemes, there are certain blocking factors that render this scheme highly inefficient for a given optimization problem (see Section 3.2: ScaLAPACK/PBLAS Routines). Because this type of inefficiency can be minimized (if not completely mitigated) based on the user's input for the blocking factors, it is imperative that a full analysis be conducted on the effect of the blocking factors on the processor idling time to determine a method for choosing optimal blocking factors to avoid this inefficiency.

5.5.1 Processor Communication Inefficiency

For the processor communication inefficiency analysis, test cases were run to log all three major and minor level functions for numbers of processors ranging from 2 to 100. To keep the test cases standardized, each test case was run with a 20-segment MSTP (see Section 5.3: Test Problem) with blocking factors MB=NB=1; the only parameter that was varied was the number of

processors. In Jumpshot-4, ten different events for each function were observed and the longest and shortest durations were tabulated. An example of one event pointing out the longest and shortest durations is shown in Figure 16. The percent difference of the longest and shortest durations with respect to the shortest duration was then calculated and averaged for all ten events of the particular function:

$$\%\ difference = \frac{longest\ duration - shortest\ duration}{shortest\ duration} * 100 \qquad (24)$$

The percent difference is a valuable measure of the program's efficiency; the larger the percent difference between the longest and shortest durations, the more time the faster processors must wait for the slowest processor to finish the event. The percent difference should be minimized in order to produce an NLP solver that is as efficient as possible.

Figure 16. Example Jumpshot-4 visualization showing longest and shortest durations for one event.

The percent differences for each function were plotted as a function of number of processors and the results are shown in Figure 17. For all three functions, as the number of processors increases, the percent difference in the longest and shortest durations for an event also increases, thereby also decreasing the overall efficiency of the code. The Hessian update function

in particular is extremely inefficient; when NLPAROPT_MPI is run on 100 processors, the percent difference in the event durations exceeds 1500%, meaning the slowest processor requires 15 times more time than the fastest processor to complete the function. In the preliminary two-processor Jumpshot-4 observation (Section 5.2: Jumpshot-4), NLPAROPT_MPI seemed relatively efficient for all three functions. However, after further analysis for larger numbers of processors, this assumption has been proven to be incorrect. Based on these results, it can be deduced that the larger number of processors require more communication time to first distribute the workload among each processor and then for one processor ("Processor 0") to receive the results from each processor once the event is complete.

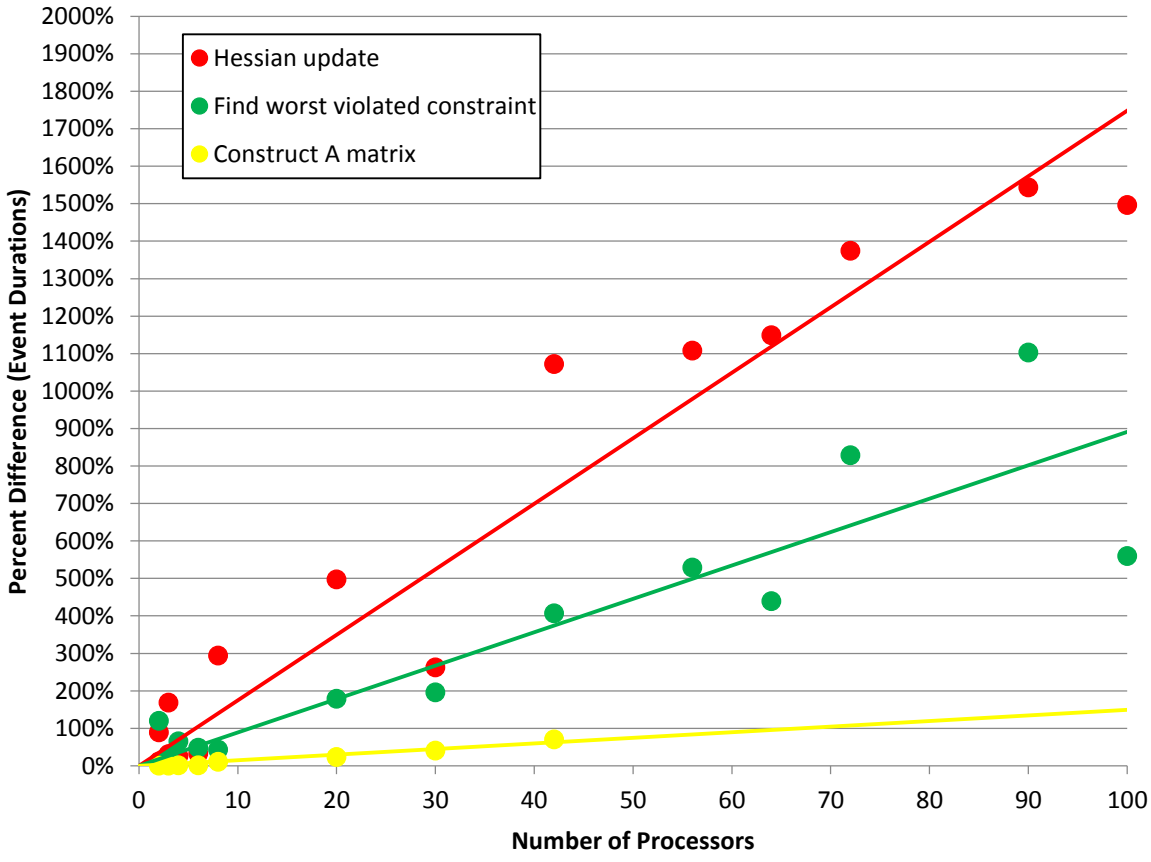


Figure 17. Percent difference of longest and shortest durations of three NLPAROPT_MPI functions versus number of processors.

This increase in communication time is simply a downside to using more processors and cannot be fully avoided, but the increase in overall runtime may be appreciable enough to offset the communication inefficiencies. To determine if and when increasing the number of processors begins to decrease the efficiency, test cases were run for 20-, 40-, 60-, 80-, and 100-segment

MSTPs (i.e. of increasing computational complexity) on 1-100 processors while keeping the blocking factor constant at MB=NB=1. The results, shown in Figure 18, indicate that there is a steady decrease in the total NLPAROPT_MPI runtime until the runtime reaches an absolute minimum; however, as the number of processors increases past this point, the total runtime increases indefinitely. This suggests that for a given optimization problem, there exists a "cross-over point" where the utilization of a processor grid containing too many processors results in a loss of efficiency due to the amount of communication overhead required for a more subdivided problem. As the computational complexity of the MSTP increases, the cross-over point occurs at a higher number of processors. Therefore, larger, more computationally complex optimization problems benefit from using more processors, whereas simpler problems should only run on a few processors to be efficient. A list of the optimal number of processors for each MSTP test case is tabulated in Table 6.

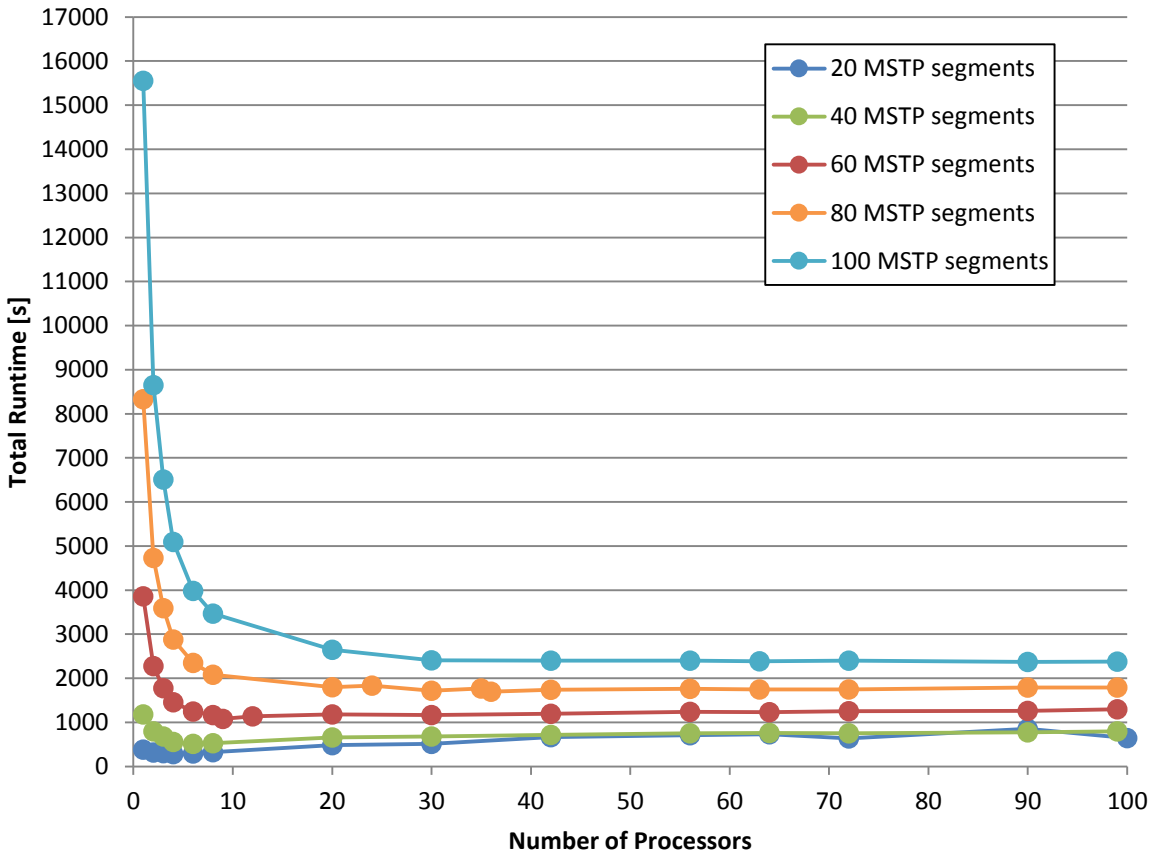


Figure 18. Total runtime as a function of number of processors for test cases with varying MSTP segments.

Table 6. Optimal number of processors for different MSTP segment test cases.

MSTP Segments	Optimal Number of Processors
20	4
40	6
60	9
80	36
100	90

For any MSTP test case, the user can now extrapolate these results to determine approximately how many processors should be used to run the problem efficiently. Although this analysis is certainly beneficial for optimizing the MSTP, most (if not all) spacecraft trajectory researchers will need to solve completely different NLPs that are unrelated to the MSTP. A universal metric for measuring a problem's computational complexity is the number of Floating Point Operations (FLOPs) that occur in the NLP. FLOPs are any mathematical computation the processor must perform, such as addition, subtraction, multiplication, and division, including operations within matrix/vector products, decompositions, and inverses [28]. The number of FLOPs for each MSTP test case can be calculated by using the following equation [29]:

$$FLOPs = \#chassis * \frac{\#nodes}{chassis} * \frac{\#sockets}{node} * \frac{\#cores}{socket} * clock * \frac{FLOPs}{cycle} * runtime \qquad (25)$$

A $clock$ is the internal clock speed of the core (2.66 GHz for Taub's Intel HP X5650 processors) and $cycle$ is defined as one clock tick. This equation can be simplified if the performance of the processor is known in FLOPs per second, or FLOPS:

$$FLOPs = FLOPS * runtime \qquad (26)$$

Taub's Intel HP X5650 processors have a FLOP performance of 124.8 GLOPS [30], and therefore the total number of FLOPs while running the MSTP test cases can be calculated by simply multiplying the FLOP performance and the total runtime of the serially-executed test case. Plotting the FLOPs versus the optimal number of processors yields a trend that is useful for any NLP optimization researcher; as long as the user knows the total number of FLOPs for the problem, he/she can use the results in Figure 19 to scale the number of processors that will allow NLPAROPT to run as efficiently as possible.

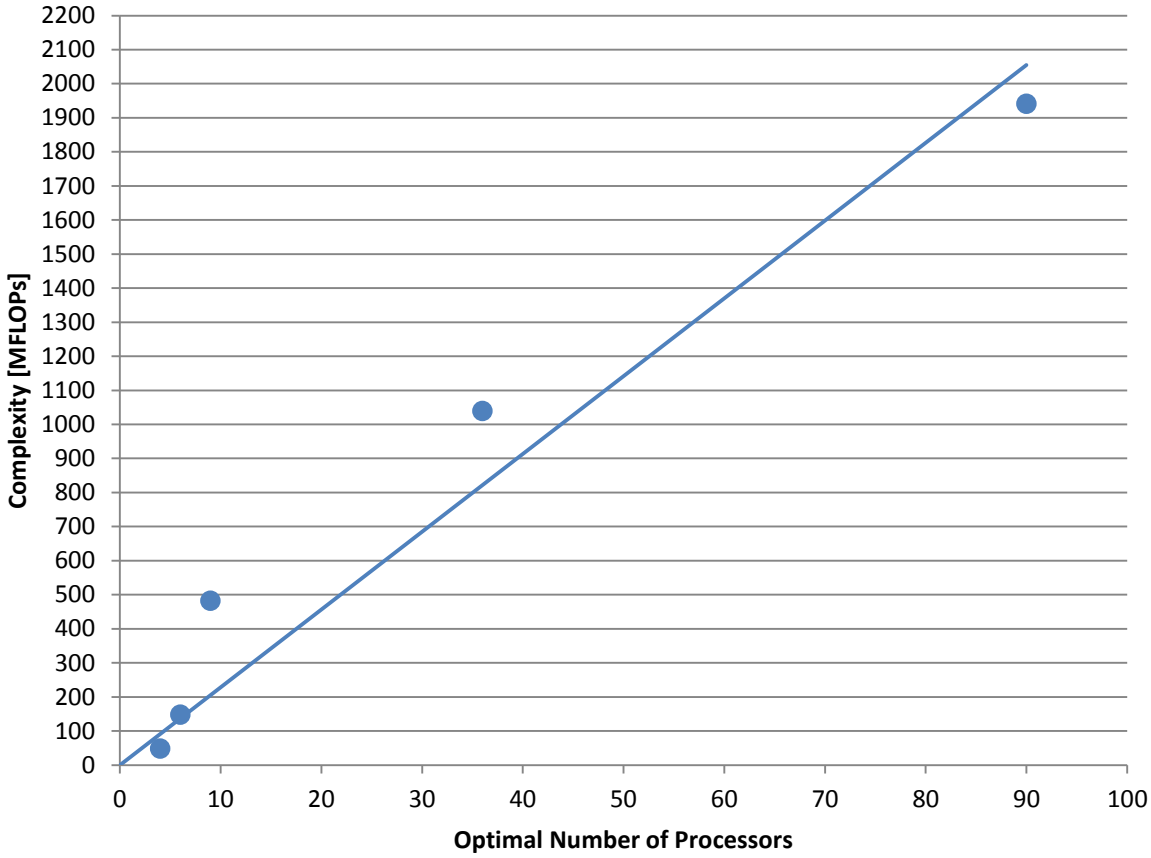
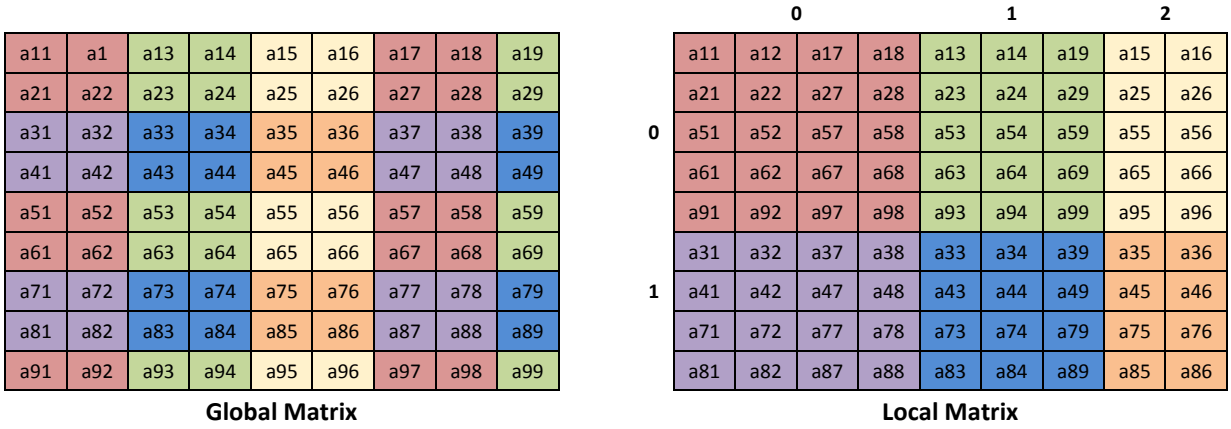


Figure 19. Computational complexity in GFLOPS versus optimal number of processors.

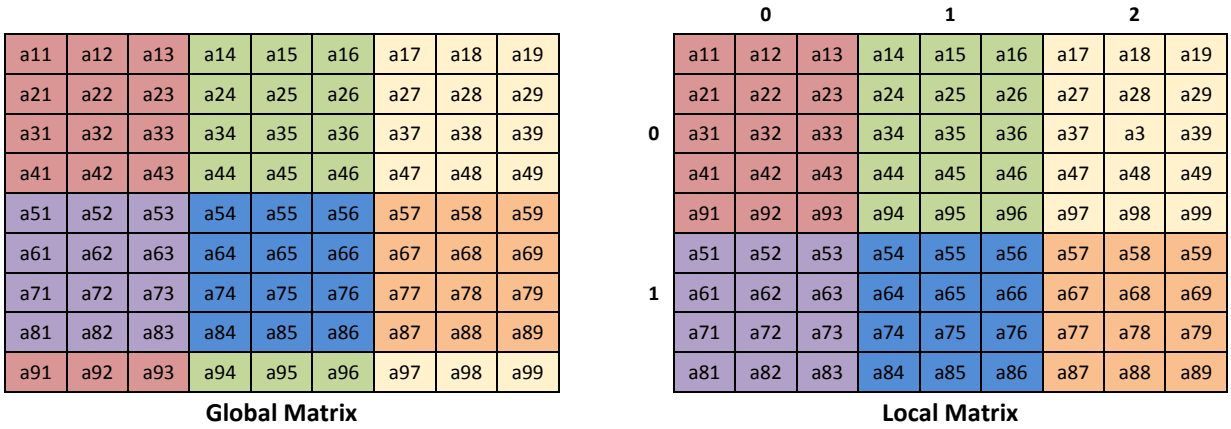
5.5.2 Uneven Workload Distribution Inefficiency

As mentioned in Section 3.2: ScaLAPACK/PBLAS Routines, the two-dimensional block-cyclic distribution scheme utilized by ScaLAPACK consistently distributes the workload among the processors as evenly as possible relative to other distribution schemes. The two-dimensional block-cyclic distribution scheme is completely dependent on the global matrix dimensions, blocking factors, and processor grid dimensions; these parameters can be optimized such that each processor receives a perfectly even workload, but conversely, certain inputs can also lead to a largely uneven distribution. In the example from Section 3.2: ScaLAPACK/PBLAS Routines, a 9x9 global matrix was distributed among six processors in a 2x3 processor grid using a square blocking factor of 2. Based on these parameters, Processor 0 receives 20 elements, while Processor 6 receives only 8; this difference in workload leads to prolonged idling times for those processors that do not have as many calculations to perform. By choosing optimal blocking factors, this difference can be decreased and in some cases eliminated. Figure 20 shows two examples

comparing the non-optimal distribution of a 9x9 matrix to a 2x3 processor grid (the example from Section 3.2: ScaLAPACK/PBLAS Routines) to an optimal distribution. In the optimal case, Processor 0 now has 15 processors, while Processor 6 has 12; this is a significantly more even distribution than the case discussed previously.



(a) Non-optimal blocking factors (MB=2, NB=2)



(b) Optimal blocking factors (MB=4, NB=3)

Figure 20. Two-dimensional block-cyclic data distribution of a 9x9 global matrix over a 2x3 processor grid using (a) non-optimal blocking factors and (b) optimal blocking factors.

For very small global matrices, it is relatively simple to determine how the matrix is distributed among the processor grid for a given blocking factor using the two-dimensional block-cyclic distribution scheme. However, from personal experience, once the global matrix dimensions exceed 10, the math becomes mind-numbing and the likelihood of errors increases in a seemingly exponential fashion. To avoid any arduous hand-calculations, a MATLAB algorithm was developed, aptly called the "Two-Dimensional Block-Cyclic Distribution Calculator" (see Appendix A.1). Given the global matrix dimensions (M and N), blocking factors (MB and NB),

and processor grid dimensions (Pr and Pc), the MATLAB script will output the number of matrix elements that are distributed to each processor.

Once this algorithm was developed and extensively tested for accuracy, a second MATLAB algorithm was developed based on the "Two-Dimensional Block-Cyclic Distribution Calculator" to vary the square blocking factors and number of processors for a given global matrix size. This new script, called the "Hessian Efficiency Plotter" (see Appendix A.2), imports a text document ("processor_grid_dimensions.txt") that lists the number of processors to plot in the first column (2 to 100 processors) and the corresponding processor grid dimensions, Pr and Pc, in the second and third columns, respectively. Though not ideal, this text document was created by hand instead of MATLAB, and the processor grid dimensions for each value for the total number of processors was calculated based on the algorithm currently implemented in NLPAROPT_MPI (shown below), which creates a processor grid to be as square as possible; it has been proven that the more square the processor grid, the more even the distribution is among the processors. This algorithm sets the jth dimension of the processor grid (Pr) to be the floor of the square root of the total number of processors ($sqrt\_procs$), and then the ith dimension is set to the largest integer that, when multiplied by Pr, is less than or equal to the total number of processors.

```
sqrt_procs = std::floor(std::sqrt(full_world.size()));
griddim_j = sqrt_procs;

for (int i = sqrt_procs; i <= full_world.size(); ++i) {

    if (sqrt_procs*i == full_world.size()) {
        griddim_i = i;
        break;
    }
    else if ((sqrt_procs*i < full_world.size()) && (full_world.size() - sqrt_procs*i < sqrt_procs)) {
        griddim_i = i;
        break;
    }
    else {
        continue;
    }
}
```

Note: This method suggests that NLPAROPT_MPI is not capable of running on certain numbers of processors that are not square or close to square. For example, attempting to run NLPAROPT_MPI on 65 processors would construct an 8x8 processor grid, therefore only running on 64 processors. Unfortunately, this is just a result of NLPAROPT_MPI still being in the preliminary design phase and a more robust algorithm will be implemented in the future to handle any number of processors.

The "Hessian Efficiency Plotter" MATLAB algorithm generates a matrix of the percent difference between the largest and smallest number of values distributed among the processors with respect to the smallest number for each number of processors (row) and square blocking factor $MB{=}NB$ (column). For example, when varying the number of processors over 99 values (between 2 and 100) and varying the square blocking factor over four values (between 2 and 5), a 99x4 matrix is generated. Using this matrix, the percent differences can be plotted as a function of total number of processors for each value of $MB{=}NB$, shown in Figure 21. .

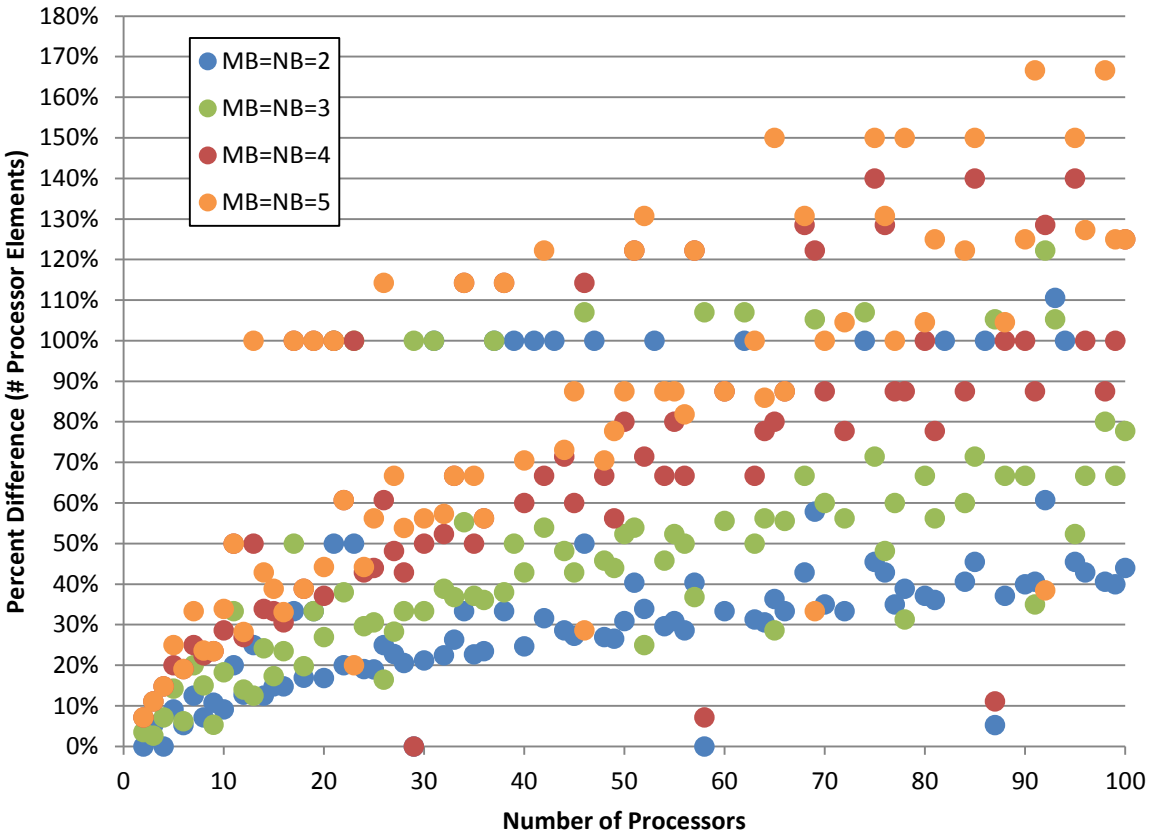


Figure 21. Percent difference of number of elements distributed among processors for the Hessian update function versus number of processors for varying square blocking factors $MB{=}NB$.

There are certain "sweet spots" where the global matrix dimensions, blocking factors, and number of processors are all chosen such that each processor receives the exact same workload. Generally, however, the results show that as the number of processors increases, the percent difference in number of elements distributed among the processors also increases. In addition, as $MB{=}NB$ increases, the percent difference also increases. From this MATLAB analysis, it can

therefore be deduced that running NLPAROPT_MPI on a larger number of processors and also increasing the blocking factor increases the likelihood that the elements are not evenly distributed

Based on this MATLAB analysis, it was hypothesized that since the percent difference in elemental distribution among the processors increases as the number of processors increases, the percent difference of the execution time for completing an event should also increase. To test this hypothesis, test cases were run on 2 to 100 processors for square blocking factors between 2 and 5. Each test case was run for a 20-segment MSTP and the results were only analyzed for the Hessian update function (Figure 22). This function was chosen among the three previously tested because the "find worst violated constraint" function does not use the ScaLAPACK library and therefore does not utilize a two-dimensional block-cyclic distribution scheme. The construction of the *A* matrix does use the ScaLAPACK library and could be used for this analysis, but the Hessian update function was chosen simply because the Hessian matrix's dimensions are half the size of the *A* matrix's dimensions and therefore the distribution is easier to visualize.

As expected, the percent difference in event completion time increases as the number of processors increases, and similar to the results for Figure 21, increasing the blocking factor also increases this percent difference. The most inefficient of these test cases is therefore the 100 processor test case with a square blocking factor of 5; the percent difference in the event duration is almost 3000%.

Though this analysis is indeed important to visualize and understand, it is even more important to develop a method of reducing these inefficiencies due to uneven processor distribution as much as possible. Given a certain number of processors and global matrix size, the distribution of the elements is purely dependent on the user's choice of the blocking factors. Therefore, the user must determine the optimal blocking factors for a given NLPAROPT_MPI problem such that the distribution of the workload among the processors is as even as possible.

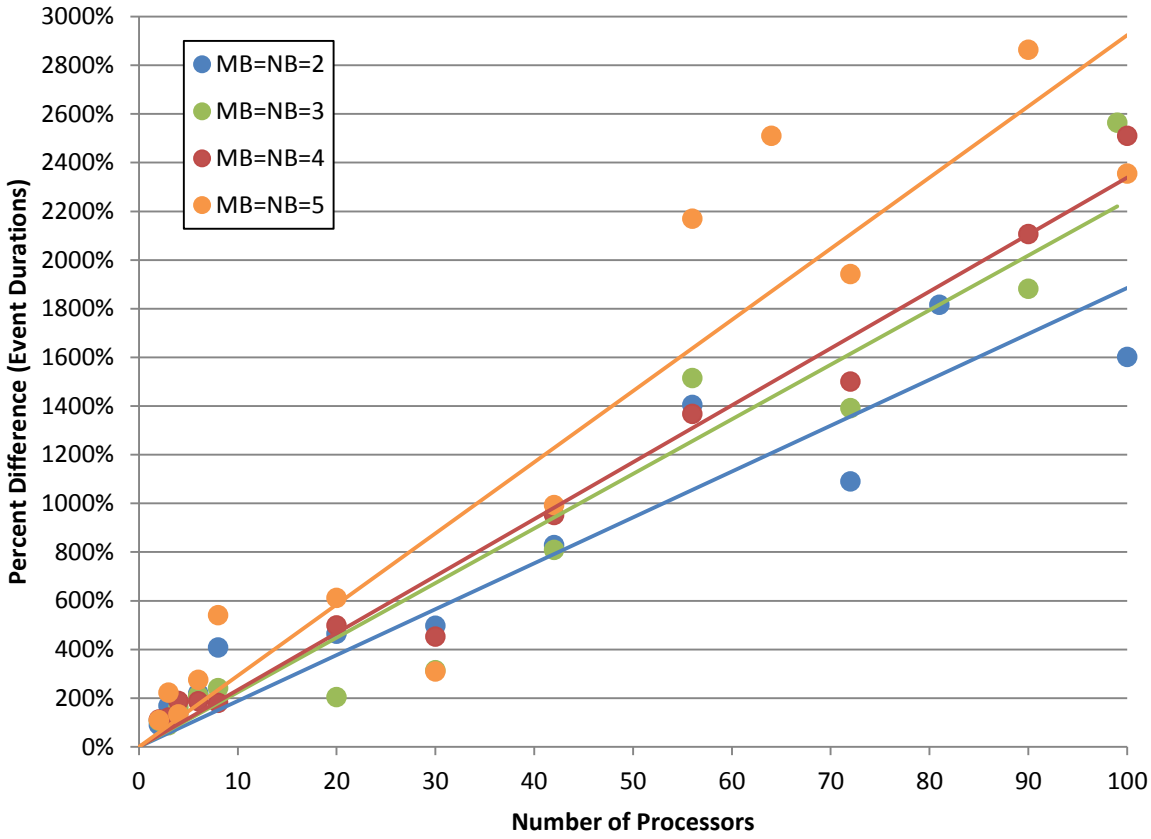


Figure 22. Percent difference of longest and shortest even durations for the Hessian update function versus number of processors for square blocking factors MB=NB between 2 and 5.

In order to determine the optimal blocking factors, a third algorithm was implemented in a MATLAB script called "Blocking Factor Optimization" (see Appendix A.3). The user inputs the global matrix dimensions (M and N) and the processor grid dimensions (Pr and Pc), and the script outputs the viable blocking factors that will yield the same minimized percent difference in the workload distribution. An example output is shown below for a 116x116 global matrix (the Hessian matrix for a 20-segment MSTP) and 6 processors in a 3x2 processor grid:

```
Smallest proc value percent difference: 2.6%
Optimal (MB,NB) values to achieve this:
1   1
3   1
13  1
1   2
3   2
13  2
1   29
3   29
13  29
1   58
3   58
13  58
```

Based on the output optimal blocking factor values, the user could choose any one of these combinations to achieve the same minimized percent difference. However, by choosing the largest optimal blocking factors (MB=13 and NB=58, in this case), the time required to parse the matrix elements among the processors is also minimized. For example, though MB=1, NB=1 result in an optimized processor distribution, these blocking factors require $ceil\left(\frac{116}{1}\right) * ceil\left(\frac{116}{1}\right) = 13{,}456$ total rounds in order to distribute all of the elements in the Hessian matrix, whereas the largest blocking factors only require $ceil\left(\frac{116}{13}\right) * ceil\left(\frac{116}{58}\right) = 18$ rounds.

After this algorithm was fully developed, the next step was to run more test cases for 2-100 processors except with optimal blocking factors instead of the arbitrary blocking factors of MB=NB=2:5 as before. Though utilizing the largest optimal blocking factors would show the greatest increase in efficiency both in terms of workload distribution and total runtime, NLPAROPT_MPI is unfortunately not yet capable of handling blocking factors that are large and not perfectly square. As previously implied, however, every test case has optimal blocking factors of MB=NB=1; though these are not the absolute optimal blocking factors, the results still show that the percent difference between the longest and shortest event durations is reduced when plotted against the other previous non-optimal blocking factors (Figure 23). The dashed line in Figure 23 therefore not only represents the efficiency trend of the optimal blocking factors of MB=NB=1, but also the upper bound of this optimal efficiency; for any number of processors, the percent difference in the event duration will be less than or equal to the values below this dashed line.

It is important to note that this optimal blocking factor analysis was only conducted on the Hessian update function. This function is indeed one of the most computationally- and time-expensive functions in NLPAROPT_MPI, but choosing optimal blocking factors that benefit the workload distribution for the Hessian update may not be optimal for other events in the program. However, this analysis yields a reasonable efficiency trend for the entire NLPAROPT_MPI program, and optimizing the blocking factors for other computationally expensive functions in the code will only further increase the efficiency.

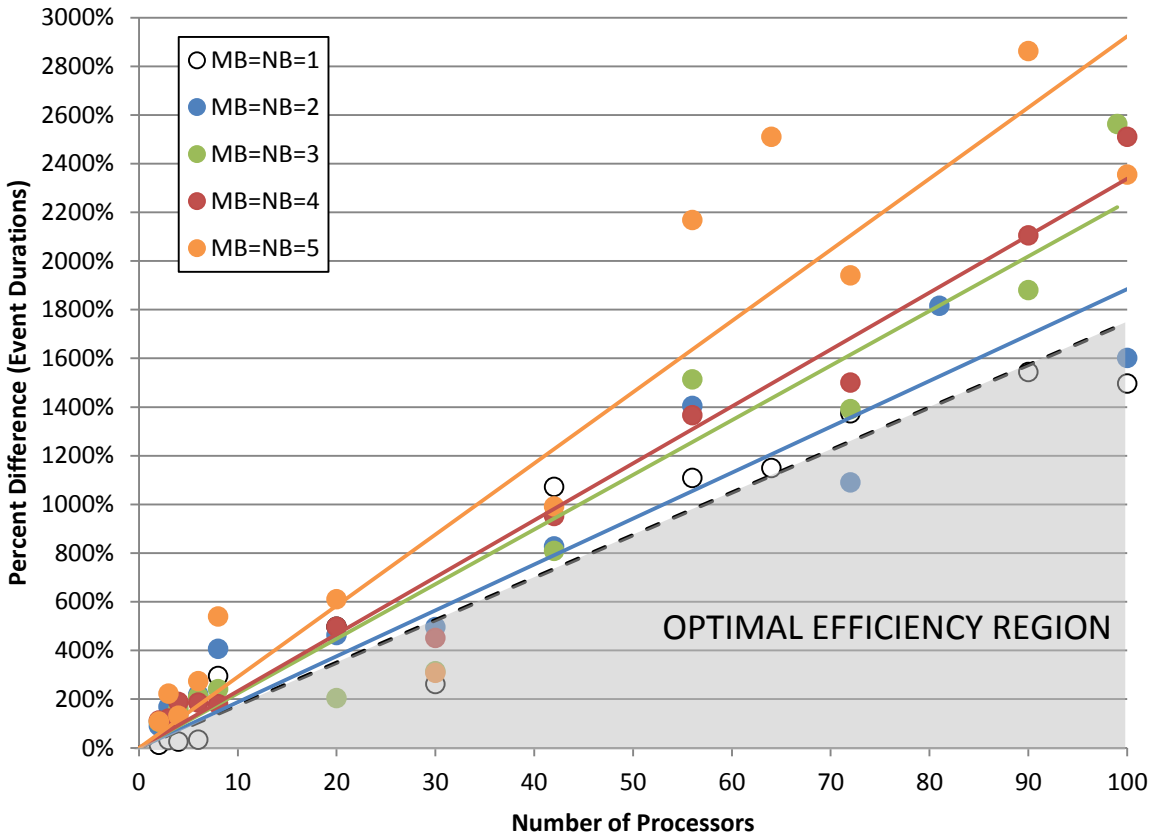


Figure 23. Percent difference of longest and shortest even durations for the Hessian update function versus number of processors for optimal and non-optimal blocking factors.

By implementing the aforementioned MATLAB algorithms to determine the optimal blocking size for the desired optimization problem, the user can expect that the processor idling time will be minimized, as well as the total program runtime, thereby potentially saving engineers, scientists, researchers, and nerdy graduate students alike a considerable amount of time, money, and sanity. Because NLPAROPT cannot yet handle large and/or non-near-square blocking factors, conducting an analysis to determine how the optimal blocking factors affect the total runtime is not yet feasible. A runtime analysis can be conducted using a blocking factor of 1, since it is technically an optimal blocking factor, but the increase in time it requires to parse out the elements using such a small blocking factor overshadows any resulting increase in runtime efficiency. Once NLPAROPT is capable of handling any blocking factor, a full analysis will be completed to show the hypothesized trend that minimizing the processor idling time results in a minimized runtime.

CHAPTER 6: FUTURE WORK

NLPAROPT has already proven itself during the Phase I effort as a promising parallel NLP solver. However, more research, development, and refining of the core algorithm, parallel implementation, and efficiency need to be conducted before being introduced to the market as a realistic competitor against the current state-of-the-art NLP solvers.

As mentioned in Section 5.5: Efficiency Analysis Results, NLPAROPT_MPI is not capable of running on certain numbers of processors and with certain blocking factors in its current state. There is still some research that needs to be conducted in order to fully understand how the Basic Linear Algebra Communication Subprogram (BLACS) operates, which is the subprogram that is responsible for producing the two-dimensional block-cyclic distribution grids [31]; once the distribution scheme algorithm is further researched, NLPAROPT_MPI can eventually be re-programmed as a robust NLP solver capable of handling any optimization problem that is thrown its way. A more in-depth efficiency analysis can then be conducted to run NLPAROPT_MPI with the largest optimal blocking factors that will (hopefully) prove the event duration percent difference is indeed minimized.

Another comparison that will certainly be of interest in the future is seeing how the implementation of optimal blocking factors other than MB=NB=1 affects the total runtime of the program. Compared to an NLPAROPT_MPI run with non-optimal blocking factors and with a number of processors that deviates from the "cross-over point" (see Section 5.5.1: Processor Communication Inefficiency), a similar run with optimal blocking factors and an optimal number of processors should undoubtedly show significant speed-ups; the question, however, is *how* significant.

The overall efficiency of NLPAROPT_MPI is only expected to become better and better as more parallelization techniques are implemented in the future. In addition, though NLPAROPT is currently not as robust as its commercial competitors, additional research and development to refine the core NLP algorithm will also help to improve the overall runtime and optimality of the solution.

CHAPTER 7: REFERENCES

[1] A. Ghosh, R. Beeson, D. Ellison, L. Richardson and J. Aurich, "NASA SBIR H9.04 Phase I Final Report for Parallel Nonlinear Optimization for Astrodynamic Navigation," 2014.

[2] J. Y. Wang, "Chapter 12 Nonlinear Programming," National Chiao Tun University Open Course Ware, 2009. [Online]. Available: http://ocw.nctu.edu.tw/upload/classbfs121001561576097.pdf. [Accessed 21 1 2014].

[3] S. P. Bradley, A. C. Hax and T. L. Magnanti, Applied Mathematical Programming, Addison-Wesley, 1977.

[4] P. E. Gill, W. Murray and M. A. Saunders, "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM Review*, vol. Vol 47, no. 1, pp. 99-131, 2005.

[5] A. Wachter, "An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering," Carnegie Mellon University, Pittsburgh, 2002.

[6] W. Karush, "Minimia of Functions of Several Variables with Inequalities as Side Constraints," Chicago, 1939.

[7] H. W. Kuhn and A. W. Tucker, "Nonlinear Programming," in *Second Berkeley on Mathematics, Statistics and Probability*, Berkeley, California, 1951.

[8] A. Peressini and J. J. J. Uhl, "The Mathematics of Nonlinear Programming," Springer, 1988.

[9] S. Boyd and L. Vandenberghe, Convex Optimization, New York: Cambridge University Press, 2004.

[10] J. Burke, "A robust trust region method for constrained nonlinear programming problems," *SIAM Journal of Optimization*, vol. 2, pp. 325-347, 1992.

[11] Y.-x. Yuan, "On the convergence of a new trust region algorithm," *Numerische Mathematik*, pp. 515-539, 1995.

[12] P. Y. Papalambros and D. J. Wilde, Principles of Optimal Design, Cambridge University Press, 2000.

[13] J. T. Betts, Practical Methods for Optimal Control Using Nonlinear Programming, Siam, 2001.

[14] A. Ghosh, R. Beeson, D. Ellison, L. Richardson and J. Aurich, "H9.04 Phase II Proposal for Parallel Nonlinear Optimization for Astrodynamic Navigation," 2014.

[15] J. Nocedal and S. J. Wright, Numerical Optimization, Berlin, New York: Springer-Verlag, 1999.

[16] J. E. Dennis, Jr. and J. J. Moré, "A Characterization of Superlinear Convergence and Its Application to Quasi-Newton Methods," *Mathematics of Computation,* vol. 28, no. 126, pp. 549-560, 1974.

[17] "LAPACK - Linear Algebra PACKage," Netlib, 16 November 2013. [Online]. Available: http://www.netlib.org/lapack/. [Accessed 5 February 2015].

[18] Netlib, September 2014. [Online]. Available: http://www.netlib.org/blas.

[19] D. Gregor and M. Troyer, "Boost C++ Libraries," Boost, 30 October 2014. [Online]. Available: http://www.boost.org/doc/libs/1_57_0/doc/html/mpi.html. [Accessed 29 1 2015].

[20] May 2009. [Online]. Available: http://netlib.org/scalapack/slug/node76.html.

[21] "Parallel Basic Linear Algebra Subprograms (PBLAS)," Netlib, [Online]. Available: http://www.netlib.org/scalapack/pblas_qref.html. [Accessed 13 12 2014].

[22] NVIDIA, "NVIDIA's Next Generation CUDA(TM) Compute Architecture: Fermi(TM)," NVIDIA, 2009.

[23] NVIDIA, "cuBLAS Library User Guide," NVIDIA, 2014.

[24] A. Chan, W. Gropp and E. Lusk, "User's Guide for MPE: Extenstions for MPI Programs," Argonne, IL.

[25] A. Chan, D. Ashton, R. Lusk and W. Gropp, "Jumpshot-4 Users Guide," Argonne, IL, 2007.

[26] A. Bryson and Y. Ho, Applied Optimal Control, Taylor and Francis, 1975.

[27] University of Illinois, "Illinois Campus Cluster Program," UIUC, 2014. [Online]. Available: https://campuscluster.illinois.edu/hardware/#taub. [Accessed 3 Feb 2015].

[28] R. Hunger, "Floating Point Operations in Matrix-Vector Calculus," Technische Universität München Associate Institute for Signal Processing, Munich, 2007.

[29] M. R. Fernandez, November 2011. [Online]. Available: http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329.

[30] Advanced Clustering Technologies, Inc., "Xeon E5 Series Overview".

[31] Netlib, 2000. [Online]. Available: http://www.netlib.org/blacs/.

[32] "A&AE 251 Fall 2005: Elementary Considerations for Rocket Design," Purdue University, 2005. [Online].

[33] A. E. Bryson, Jr. and Y.-C. Ho, Applied Optimal Control, CRC Press, 1975.

[34] S. Blackford, "The Two-Dimensional Block-Cyclic Distribution," Netlib, 13 May 2009. [Online]. Available: http://netlib.org/scalapack/slug/node75.html. [Accessed 12 13 2014].

[35] "About Blue Waters," National Center for Supercomputing Applications, 2014. [Online]. Available: http://www.ncsa.illinois.edu/enabling/bluewaters. [Accessed 3 Feb 2015].

[36] February 2005. [Online]. Available: http://scv.bu.edu/~kadin/PACS/numlib/chapter2/dist.gif.

APPENDIX: MATLAB SCRIPTS

A.1 TWO-DIMENSIONAL BLOCK-CYCLIC DISTRIBUTION CALCULATOR

```
% Two-Dimensional Block-Cyclic Distribution Calculator
% Laura Richardson, 3/18/2015

% This MATLAB script serves to calculate the distribution of values in a
% given M x N matrix with blocking factors MB,NB among a number of
% processors in a Pr x Pc processor grid using a two-dimensional block-cyclic
% distribution.

clear all
close all
clc

% USER INPUTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M = 116;  % global matrix row dimension
N = 116;  % global matrix column dimension

MB = 2;  % row-wise blocking factor
NB = 2;  % column-wise blocking factor

Pr = 3;  % number of rows in processor grid
Pc = 2;  % number of columns in processor grid

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

total_procs = Pr*Pc;

M_blocks = floor(M/MB);  % number of whole MB blocks M is split up into
N_blocks = floor(N/NB);  % number of whole NB blocks N is split up into

% the "block grid" (M_blocks x N_blocks) is now a subset of the global
% matrix that is only consumed by full blocks, i.e. has removed any excess
% hanging rows or columns on the ends that do not make up a complete block

extra_blocks_row = M-M_blocks*MB;  % number of hanging rows
extra_blocks_col = N-N_blocks*NB;  % number of hanging columns

source_location = [0 0];  % location of 0th processor in processor grid

for i = 1:Pr
    for j = 1:Pc

        proc_number = j+Pc*(i-1);

        my_location = [i-1 j-1];  % my location in the proc grid

        % determine the distance between me and the source
        % processor (0,0) in the processor grid
        my_dist_row = mod(Pr+my_location(1)-source_location(1),Pr);
```

```
my_dist_col = mod(Pc+my_location(2)-source_location(2),Pc);

% determine the number of rows and columns that each processor has
% in ONLY the block grid:

% if the block grid is divided evenly among the processors
% row-wise...
if mod(M_blocks,Pr) == 0
    num_row = (M_blocks/Pr)*MB;

% otherwise if my processor receives more rows than other
% processors...
elseif mod(MB*M_blocks,MB*Pr)/MB-1 >= my_dist_row
    num_row = mod(MB*M_blocks,MB*Pr)/(mod(MB*M_blocks,MB*Pr)/MB)+...
              (MB*M_blocks-mod(MB*M_blocks,MB*Pr))/Pr;

% otherwise if my processor simply gets the minimum amount of full
% blocks in the block grid...
else
    num_row = (MB*M_blocks-mod(MB*M_blocks,MB*Pr))/Pr;
end

% if the block grid is divided evenly among the processors
% column-wise...
if mod(N_blocks,Pc) == 0
    num_col = (N_blocks/Pc)*NB;

% otherwise if my processor receives more rows than other
% processors...
elseif mod(NB*N_blocks,NB*Pc)/NB-1 >= my_dist_col
    num_col = mod(NB*N_blocks,NB*Pc)/(mod(NB*N_blocks,NB*Pc)/NB)+...
              (NB*N_blocks-mod(NB*N_blocks,NB*Pc))/Pc;

% otherwise if my processor simply gets the minimum amount of full
% blocks in the block grid...
else
    num_col = (NB*N_blocks-mod(NB*N_blocks,NB*Pc))/Pc;
end

% if there are hanging rows/columns, determine which processor
% receives the values in the extra rows/columns:

if extra_blocks_row > 0
    if my_dist_row == mod(M_blocks,Pr)
        num_row = num_row+extra_blocks_row;
    else
        num_row = num_row;
    end
end

if extra_blocks_col > 0
    if my_dist_col == mod(N_blocks,Pc)
        num_col = num_col+extra_blocks_col;
    else
        num_col = num_col;
```

```
            end
        end

        total_values = num_row*num_col;
        proc_values(proc_number) = total_values;
        total_matrix_values = M*N;
        fprintf('Processor %i values: %i\n',proc_number,total_values)

    end
end

min_proc_value = min(proc_values);
max_proc_value = max(proc_values);
sum_proc_values = sum(proc_values);

if sum_proc_values ~= total_matrix_values
    fprintf('Number of values for each processor does not add up to the full
             matrix dimensions. Something is wrong.')
end

percent_difference = ((max_proc_value-min_proc_value)/min_proc_value)*100;
fprintf('Min processor value: %i\n',min_proc_value)
fprintf('Max processor value: %i\n',max_proc_value)
```

A.2 HESSIAN EFFICIENCY PLOTTER

```
% Hessian Efficiency Plotter
% Laura Richardson, 3/24/2015

% This MATLAB script stems off of the "Two-Dimensional Block-Cyclic
% Distribution Calculator" script and utilizes the same logic to calculate
% the number of values each processor is distributed for a given matrix size
% (M x N), blocking factors (MB, NB), and processor grid (Pr x Pc).

% However, rather than the user specifying the desired blocking factors and
% processor grid dimensions, this script instead loops over the desired
% values of MB/NB as well as the number of processors (and subsequently,
% the processor grid). A plot is then generated for the number of
% processors vs. the percent difference in the values distributed among
% each processor for different values of MB/NB.

clear all
close all
clc

% USER INPUTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M = 116;  % global matrix row dimension
N = 116;  % global matrix column dimension

processor_grid_dimensions = importdata('processor_grid_dimensions.txt');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for MB = 2:5

    NB = MB;
    proc_values = [];

    for k = 1:length(processor_grid_dimensions)

        total_procs = processor_grid_dimensions(k,1);
        Pr = processor_grid_dimensions(k,2);
        Pc = processor_grid_dimensions(k,3);

        M_blocks = floor(M/MB);  % number of whole MB blocks M is split up
                                   into
        N_blocks = floor(N/NB);  % number of whole NB blocks N is split up
                                   into
        % the "block grid" (M_blocks x N_blocks) is now a subset of the
        % global matrix that is only consumed by full blocks, i.e. has
        % removed any excess hanging rows or columns on the ends that do not
        % make up a complete block

        extra_blocks_row = M-M_blocks*MB;  % number of hanging rows
        extra_blocks_col = N-N_blocks*NB;  % number of hanging columns
```

```
source_location = [0 0];  % location of 0th processor in processor
                          grid

    for i = 1:Pr
    for j = 1:Pc

        proc_number = j+Pc*(i-1);

        my_location = [i-1 j-1];  % my location in the proc grid

        % determine the distance between me and the source
        % processor (0,0) in the processor grid
        my_dist_row = mod(Pr+my_location(1)-source_location(1),Pr);
        my_dist_col = mod(Pc+my_location(2)-source_location(2),Pc);

        % determine the number of rows and columns that each
        % processor has in ONLY the block grid:

        % if the block grid is divided evenly among the processors
        % row-wise...
        if mod(M_blocks,Pr) == 0
            num_row = (M_blocks/Pr)*MB;

        % otherwise if my processor receives more rows than other
        % processors...
        elseif mod(MB*M_blocks,MB*Pr)/MB-1 >= my_dist_row
            num_row = mod(MB*M_blocks,MB*Pr)/...
                      (mod(MB*M_blocks,MB*Pr)/MB)+...
                      (MB*M_blocks-mod(MB*M_blocks,MB*Pr))/Pr;

        % otherwise if my processor simply gets the minimum amount of
        % full blocks in the block grid...
        else
            num_row = (MB*M_blocks-mod(MB*M_blocks,MB*Pr))/Pr;
        end

        % if the block grid is divided evenly among the processors
        % column-wise...
        if mod(N_blocks,Pc) == 0
            num_col = (N_blocks/Pc)*NB;

        % otherwise if my processor receives more rows than other
        % processors...
        elseif mod(NB*N_blocks,NB*Pc)/NB-1 >= my_dist_col
            num_col = mod(NB*N_blocks,NB*Pc)/...
                      (mod(NB*N_blocks,NB*Pc)/NB)+...
                      (NB*N_blocks-mod(NB*N_blocks,NB*Pc))/Pc;

        % otherwise if my processor simply gets the minimum amount of
        % full blocks in the block grid...
        else
            num_col = (NB*N_blocks-mod(NB*N_blocks,NB*Pc))/Pc;
        end
```

```
                % if there are hanging rows/columns, determine which
                % processor receives the values in the extra rows/columns:

                if extra_blocks_row > 0
                    if my_dist_row == mod(M_blocks,Pr)
                        num_row = num_row+extra_blocks_row;
                    else
                        num_row = num_row;
                    end
                end

                if extra_blocks_col > 0
                    if my_dist_col == mod(N_blocks,Pc)
                        num_col = num_col+extra_blocks_col;
                    else
                        num_col = num_col;
                    end
                end

                total_values = num_row*num_col;
                proc_values(proc_number) = total_values;
                total_matrix_values = M*N;

            end
        end

        min_proc_value = min(proc_values);
        max_proc_value = max(proc_values);
        sum_proc_values = sum(proc_values);

        if sum_proc_values ~= total_matrix_values
            fprintf('Number of values for each processor does not add up to
                    the full matrix dimensions. Something is wrong.')
        end

        percent_difference(k,MB) = ((max_proc_value-min_proc_value)/...
                                    min_proc_value)*100;

    end
end

figure(1)
plot(processor_grid_dimensions(:,1),percent_difference(:,2),'*')
xlabel('# processors')
ylabel('% difference')
title('MB=NB=2')
ylim([0 200])

figure(2)
plot(processor_grid_dimensions(:,1),percent_difference(:,3),'*')
xlabel('# processors')
ylabel('% difference')
title('MB=NB=3')
ylim([0 200])
```

```
figure(3)
plot(processor_grid_dimensions(:,1),percent_difference(:,4),'*')
xlabel('# processors')
ylabel('% difference')
title('MB=NB=4')
ylim([0 200])

figure(4)
plot(processor_grid_dimensions(:,1),percent_difference(:,5),'*')
xlabel('# processors')
ylabel('% difference')
title('MB=NB=5')
ylim([0 200])
```

A.3 BLOCKING FACTOR OPTIMIZATION

```
% Blocking Factor Optimization
% Laura Richardson, 3/18/2015

% This MATLAB script stems off of the "Two-Dimensional Block-Cyclic
% Distribution Calculator" script and utilizes the same logic to calculate
% the number of values each processor is distributed for a given matrix size
% (M x N), blocking factors (MB, NB), and processor grid (Pr x Pc).

% This script determines the optimal blocking factors to achieve the
% smallest percent difference in the distribution of values among the
% processors given a global matrix (M x N) and processor grid (Pr x Pc).

clear all
close all
clc

% USER INPUTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M = 116;  % global matrix row dimension
N = 116;  % global matrix column dimension

Pr = 3;  % number of rows in processor grid
Pc = 2;  % number of columns in processor grid

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

total_procs = Pr*Pc;

for MB = 1:floor(M/Pr)
    for NB = 1:floor(N/Pc)

        M_blocks = floor(M/MB);  % number of whole MB blocks M is split up
                                   into
        N_blocks = floor(N/NB);  % number of whole NB blocks N is split up
                                   into

        % the "block grid" (M_blocks x N_blocks) is now a subset of the
        % global matrix that is only consumed by full blocks, i.e. has
        % removed any excess hanging rows or columns on the ends that do not
        % make up a complete block

        extra_blocks_row = M - M_blocks*MB;  % number of hanging rows
        extra_blocks_col = N - N_blocks*NB;  % number of hanging columns

        source_location = [0 0];  % location of 0th processor in processor
                                    grid

        for i = 1:Pr
            for j = 1:Pc

                proc_number = j+Pc*(i-1);
```

```
my_location = [i-1 j-1];  % my location in the proc grid

% determine the distance between me and the source
% processor (0,0) in the processor grid
my_dist_row = mod(Pr+my_location(1)-source_location(1),Pr);
my_dist_col = mod(Pc+my_location(2)-source_location(2),Pc);

% determine the number of rows and columns that each
% processor has in ONLY the block grid:

% if the block grid is divided evenly among the processors
% row-wise...
if mod(M_blocks,Pr) == 0
    num_row = (M_blocks/Pr)*MB;

% otherwise if my processor receives more rows than other
% processors...
elseif mod(MB*M_blocks,MB*Pr)/MB-1 >= my_dist_row
    num_row = mod(MB*M_blocks,MB*Pr)/...
              (mod(MB*M_blocks,MB*Pr)/MB)+...
              (MB*M_blocks-mod(MB*M_blocks,MB*Pr))/Pr;

% otherwise if my processor simply gets the minimum
% amount of full blocks in the block grid...
else
    num_row = (MB*M_blocks-mod(MB*M_blocks,MB*Pr))/Pr;
end

% if the block grid is divided evenly among the processors
% column-wise...
if mod(N_blocks,Pc) == 0
    num_col = (N_blocks/Pc)*NB;

% otherwise if my processor receives more rows than other
% processors...
elseif mod(NB*N_blocks,NB*Pc)/NB - 1 >= my_dist_col
    num_col = mod(NB*N_blocks,NB*Pc)/...
              (mod(NB*N_blocks,NB*Pc)/NB)+...
               (NB*N_blocks-mod(NB*N_blocks,NB*Pc))/Pc;

    % otherwise if my processor simply gets the minimum
    % amount of full blocks in the block grid...
else
    num_col = (NB*N_blocks-mod(NB*N_blocks,NB*Pc))/Pc;
end

% if there are hanging rows/columns, determine which
% processor receives the values in the extra rows/columns:

if extra_blocks_row > 0
    if my_dist_row == mod(M_blocks,Pr)
        num_row = num_row+extra_blocks_row;
    else
        num_row = num_row;
```

```
                end
            end

            if extra_blocks_col > 0
                if my_dist_col == mod(N_blocks,Pc)
                    num_col = num_col+extra_blocks_col;
                else
                    num_col = num_col;
                end
            end

            % store the number of values belonging to each processor in a
            % vector, and print out how many values each processor has:
            total_values = num_row*num_col;
            proc_values(proc_number) = total_values;
            total_matrix_values = M*N;

        end
    end

    min_proc_value = min(proc_values);
    max_proc_value = max(proc_values);
    sum_proc_values = sum(proc_values);

    if sum_proc_values ~= total_matrix_values
        fprintf('Number of values for each processor does not add up to
                 the full matrix dimensions. Something is wrong.')
    end

    percent_difference(MB,NB) = ((max_proc_value-min_proc_value)/...
                                 min_proc_value)*100;

  end
end

min_percent_difference = min(min(percent_difference));
[r,c] = find(percent_difference == min_percent_difference);
optimal_MB_NB_factors = [r,c];

fprintf('Smallest proc value percent difference: %0.1f%%\n',
        min_percent_difference)
fprintf('Optimal (MB,NB) values to achieve this:\n')

for i = 1:length(r)
    fprintf('%i\t%i\t\n', r(i), c(i))
end
```