MULTILEVEL PARALLELISM FOR OPTIMIZATION ON MP COMPUTERS: THEORY AND EXPERIMENT

M. S. Eldred*, W. E. Hart†, B. D. Schimel‡, and B. G. van Bloemen Waanders§
Optimization and Uncertainty Estimation Department
Sandia National Laboratories¶
Albuquerque, NM 87185

Abstract

Parallel optimization approaches which exploit only a single type of parallelism (e.g., a single simulation instance executes in parallel or an optimization algorithm manages concurrent serial analyses) have clear performance limitations that prevent effective scaling with the thousands of processors available in massively parallel (MP) supercomputers. This motivated the development of a two-level parallelism capability in which concurrent instances of multiprocessor simulations are coordinated. The most effective processor partitioning scheme in this case is the one that limits the size of parallel simulations in favor of concurrent execution of multiple simulations. That is, if both coarse-grained and fine-grained parallelism can be exploited, then preference should be given to the coarse-grained parallelism.

In this paper, two-level parallelism results are extended to the case of an arbitrary number of levels in order to maximize coarse-grained concurrency and achieve improved scaling on MP computers. An extended mathematical analysis results in recommended processor partitioning schemes which maximize parallel work, thereby minimizing efficiency losses due to communication and scheduling. Experimental validation of these results is presented for up to four nested levels of parallelism. While the two-level result is simple and intuitive, the multilevel recommendations for selecting coarse-grained concurrency among multiple levels can vary depending on scheduling approach and simulation duration heterogeneity.

*Principal Member of Technical Staff, Mail Stop 0847, AIAA senior member.
†Principal Member of Technical Staff, Mail Stop 1110.
‡Postdoctoral appointee, Mail Stop 0847.
§Principal Member of Technical Staff, Mail Stop 0847.
¶Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

Introduction

Parallel computers within the Department of Energy national laboratories have exceeded three trillion floating point operations per second (3 TeraFLOPS, theoretical peak) and are expected to achieve 100 TeraFLOPS by 2004. This performance is achieved through the use of massively parallel processing (O[10^3-10^4] processors). In order to harness the power of these machines for performing design, parallel optimization approaches are needed which are scalable on thousands of processors. To better understand the possibilities, it is instructive to first categorize the opportunities for exploiting parallelism in optimization into four main areas [1] consisting of coarse-grained and fine-grained parallelism opportunities within algorithms and their function evaluations:

- *Algorithmic coarse-grained parallelism*: This parallelism involves the concurrent execution of multiple independent function evaluations, where a function evaluation is defined as a data request from an algorithm (which may involve multiple objective and constraint evaluations). This parallelism can be exploited in gradient-based algorithms (e.g., finite differencing, speculative optimization), nongradient-based algorithms (e.g., genetic algorithms, parallel direct search), and approximate methods (e.g., design of computer experiments for building response surfaces). This concept can also be extended to the concurrent execution of multiple "iterators" (e.g., optimization, uncertainty quantification) within a "strategy" (e.g., branch and bound, optimization under uncertainty, collaborative/concurrent subspace formulations for MDO [2]).
- *Algorithmic fine-grained parallelism:* This involves computing the basic computational steps of an optimization algorithm (i.e., the internal linear algebra) in parallel. This is primarily of interest in large-scale optimization problems and simultaneous analysis and design (SAND) [3].
- *Function evaluation coarse-grained parallelism:* This involves concurrent computation of separable parts of a single function evaluation. This parallelism can be exploited when the evaluation of objective and constraint functions requires multiple independent simulations (e.g. multiple loading or operational

environments) or multiple dependent analyses where the coupling is applied at the optimizer level (e.g., the individual discipline feasible formulation [4]).

- *Function evaluation fine-grained parallelism:* This involves parallelization of the solution steps within a single analysis code. Sandia has developed MP codes in the areas of nonlinear mechanics, structural dynamics, chemically-reacting flows, thermal mechanics, shock physics, and many others.

For similar classifications of parallelism sources in optimization, refer to [5] and [6]. By definition, coarse-grained parallelism requires very little inter-processor communication and is therefore “embarrassingly parallel,” meaning that there is little loss in parallel efficiency due to communication as the number of processors increases. However, it is often the case that there are not enough separable computations on each optimization cycle to utilize the thousands of processors available on MP machines. This limitation was shown in a parallel coordinate pattern search optimization in which the maximum speedup exploiting *only* coarse-grained algorithmic parallelism was shown to be severely limited by the size of the design problem [7] (coordinate pattern search has at most $2n$ independent evaluations per cycle for n design variables).

Fine-grained parallelism, on the other hand, involves much more communication among processors and care must be taken to avoid the case of inefficient machine utilization in which the communication demands among processors outstrip the amount of actual computational work to be performed. This limitation was illustrated for a chemically-reacting flow simulation of fixed size in which it is shown that, while simulation run time does monotonically decrease with increasing number of processors, the relative parallel efficiency $\hat{E}$ of the computation for fixed model size is decreasing rapidly [1] (from $\hat{E} = 0.87$ at 64 processors to $\hat{E} = 0.39$ at 512 processors). This is due to the fact that the total amount of computation is approximately fixed, whereas the communication demands are increasing rapidly with increasing numbers of processors. Therefore, there is an effective limit on the number of processors that can be employed for fine-grained parallel simulation of a particular model size, and only for extreme model sizes (“heroic-scale”) can thousands of processors be efficiently utilized in optimization exploiting fine-grained parallelism alone.

These limitations point us to the exploitation of multiple levels of parallelism, in particular the combination of coarse-grained and fine-grained approaches. This concept is not without precedent. Two and three level parallel implementations are described in [8] and [9] for chemically reacting flow and aeroelastic optimizations, respectively.

If multiple types of parallelism can be exploited, the question arises: how should the amount of parallelism at each level be selected so as to maximize the overall efficiency of the study? For two levels of parallelism, the relative parallel efficiency $\hat{E}$ of a two-level parallel optimizer on p processors, as a function of the size p' of multiprocessor simulations within p, is:

$$\hat{E}(p') \cong \frac{C}{p'\bar{T}(p')} \quad \text{for some constant } C > 0$$

where $\bar{T}(p')$ is the time required to run a multiprocessor simulation [1]. For sublinear analysis speedup (the predominating case), the denominator is monotonically increasing with p' and $\hat{E}(p')$ is maximized at $p' = p'_{min}$, where p'_{min} is the minimum number of processors on which a parallel simulation can execute. Therefore, the size of fine-grained parallel simulations is to be minimized by exploiting as much coarse-grained parallelism as possible. Important practical considerations and caveats are discussed in [10]. Computational evidence for this conclusion is provided in [1] by demonstrating minimum run time for $p' = p'_{min}$ on clusters of workstations for the case of a simple test simulator.

A simplified depiction of the effect on scalability is given in Figure 1 in which the limitations of single-level parallelism are shown along with the effect of combining the approaches into multilevel parallelism. By minimizing p', we move as far back on the fine-grained parallelism curve as possible, into the near-linear scaling range. And then we replicate this fine-grained parallel performance with multiple coarse-grained instances.

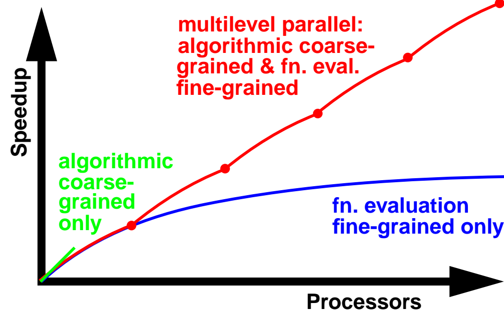


Figure 1. Scalability for fixed model size.

These insights lead naturally to the desire to maximize the available coarse-grained instances, since this allows the replication of yet more linear fine-grained parallel performance and therefore improved overall scaling. We seek to maximize concurrency in this way through the nesting of multiple parallelism levels. The following sections present an extended analysis for an arbitrary number of levels, followed by implementation, computational experiments, and concluding discussion.

Theory

This section analyzes the parallel performance of an abstract multilevel optimization algorithm (MOA). This analysis extends and refines our earlier analysis of two-level parallel optimizers [1]. We consider a MOA that has k levels of parallelism, which has $k-1$ levels that use a fixed scheduling policy to distribute work to processor partitions, and a final level of processor partitions that run an analysis code. Our analysis will be particularly concerned with multilevel algorithms for which the $k-1$ levels define a parallel optimization process and the last level is an expensive, parallel algorithm (e.g. a parallel simulation) whose solution time dominates the computational costs of the entire MOA.

To define a MOA, let A_i refer to the algorithm running at level i, and let τ_i be the number of independent servers used by A_i. Thus τ_i defines the number of partitions of processors that A_i communicates with (in addition to its master process, if it has one). If an algorithm at level i is given ${p'}_i$ processors, then it uses $m_i \in \{0,1\}$ processors for a master process and gives each partition at least $\lfloor ({p'}_i - m_i)/\tau_i \rfloor$ processors. The remaining processors are distributed amongst the partitions arbitrarily so that the maximum number of processors in a partition is $\lceil ({p'}_i - m_i)/\tau_i \rceil$.

A standard measure of parallel performance is *efficiency*. The efficiency of a parallel algorithm is $E(p) = T(1)/(pT(p))$, where $T(p)$ is the time required by the algorithm on p processors, and $T(1)$ is the time required by the best known serial version of the algorithm. When a serial version of an algorithm is not available or when a problem is too large to be run on a single processor (e.g. in terms of memory requirements), an alternative measure of parallel performance is *relative efficiency*. The relative efficiency of an algorithm is

$$\hat{E}(p) = \frac{E(p)}{E(p_{\min})} = \frac{p_{\min} T(p_{\min})}{pT(p)},$$

where $p_{\min}$ is the smallest number of processors to which the algorithm can be applied.

Note that $p_{\min}$ is typically large for engineering applications at Sandia, where the lowest level of parallelism is a parallel numerical simulation for which the number of processors is constrained by the size of the dataset. Thus we focus on the relative efficiency of MOAs. We assume without loss of generality that a judicious choice of τ_i has been made to ensure that the MOA can fit onto p processors. The numerator of $\hat{E}(p)$ is fixed, so maximizing relative efficiency is equivalent to minimizing the parallel work, $W = pT(p)$, of a parallel algorithm.

The parallel work of a multilevel algorithm includes the work of each algorithm A_i as well as the work of scheduling algorithms used to distribute work to the processor partitions at each level. The generic nature of MOAs makes it extremely difficult to provide an exact performance analysis that is valid for a broad range of MOAs. Instead, we make the following simplifying assumptions that enable us to provide an analysis of a class of algorithms that captures salient features of most MOAs:

Communication: When a self-scheduling or distributed scheduling policy (see Section *Scheduling Within Levels*) is used to distribute work to the servers for A_i, communication is needed to send the results back and forth between A_{i+1}. We assume that the time required to communicate between A_i and A_{i+1} is a fixed constant T_i^{comm}, independent of the size of the message or the underlying network topology. These communication times are the same for all A_i.

Serial Work: We assume that the time required to execute the serial work of A_i, T_i^{serial} is constant for all problems.

Concurrency: We assume that the parallelism in A_i is the same regardless the task that it is computing. We can model the parallelism in A_i in a generic fashion by denoting the number of concurrent calls to worker processes that are used by A_i. Let $n_{i,j}$ denote the number of concurrent calls to worker processes for A_i in the j-th phase of the algorithm, $j = 1, \ldots, N_i$. This assumption means that the values N_i and n_{ij} are the same every time A_i is run.

Given these assumptions, the time required for a master process at the i-th level of an MOA to run on p' processors is

$$T_i(p') = T_i^{\text{serial}} + \gamma_i 2T_i^{\text{comm}} + \gamma_i T_{i+1}(\lfloor (p' - m_i)/\tau_i \rfloor),$$

where $\gamma_i = \sum_{j=1}^{N_i} \lceil n_{ij}/\tau_i \rceil$, which denotes how many problems need to be computed in parallel on the A_{i+1} servers in order for A_i to solve a problem. For $k > 1$ it follows that

$$\begin{aligned} T(p) &= \sum_{i=1}^{k-1} \Pi_{j=0}^{i-1} \gamma_j T_i^{\text{serial}} + \sum_{i=1}^{k-1} 2T_i^{\text{comm}} \Pi_{j=1}^{i} \gamma_j \\ &\quad + T_k(p'_k) \Pi_{i=1}^{k-1} \gamma_i, \end{aligned}$$

where $\gamma_0 = 1$, $p'_1 = p$ and $p'_k = \lfloor (p'_{k-1} - m_{k-1})/\tau_{k-1} \rfloor$.

A general evaluation of $W(p)$ is not possible because of the dependence of $T(p)$ on N_i and n_{ij}, which are not generally known. However, for given values of the γ_i it is possible to evaluate the relative utility of different MOA. In particular, we consider the choice

of the τ_i variables for two scheduling policies: static scheduling and self-scheduling. For self-scheduling, we assume that a master process is used to execute A_i and A_{i+1} is executed on the processor partitions. If a static scheduling policy is used, then the execution of A_i is replicated across all processors and no master process is used. Since our analysis does not consider the case where the execution times for the A_i may differ, the difference between these two cases in the following theoretical analysis are simply reflected by whether or not a master process is used (i.e. by the value of m_i).

We assume that the work for A_k dominates the computation of the MOA, so the first two terms in $T(p)$ are small compared to the last term of $T(p)$. Further, in practice we can choose τ_i such that $p'_k \approx p/\Pi_{i=1}^{k-1}\tau_i$. Thus we have

$$W(p) \approx pT_k(p/\Pi_{i=1}^{k-1}\tau_i)\Pi_{i=1}^{k-1}\gamma_i.$$

The optimal values for the τ_i are very dependent upon how the γ_i vary with the τ_i. For example, if there is a wide range of n_{ij} for A_i, then the optimal value of τ_i will depend upon the characteristics of the distribution of n_{ij}. However, we have observed that a common case is when the n_{ij} are constant with respect to j. In this case, $n_{ij} = n_i$ so $\gamma_i = N_i \lceil n_i/\tau_i \rceil$.

To simplify our analysis, we constrain ourselves to solutions where n_i is an integer multiple of τ_i (which is commonly done in practice). If $w_i = n_i/\tau_i$, then the parallel work can be rewritten as

$$W(p) \approx C\rho T_k(\rho),$$

where $\rho = p/\Pi_{i=1}^{k-1}\tau_i$ and $C = \Pi_{i=1}^{k-1}N_i n_i$. Note that

$$p_{\min}^i = m_i + \tau_i p_{\min}^{i+1},$$

where $p_{\min}^i$ is the minimum number of processors required for A_i. It follows that

$$p_{\min} = p_{\min}^k \Pi_{i=1}^{k-1}\tau_i + \sum_{i=1}^{k-2} m_{i+1}\Pi_{j=1}^{i}\tau_j + m_1.$$

Consequently, we can minimize the parallel work for a given p by finding $\tau_1, \ldots, \tau_k$ that solve the problem:

$$\begin{array}{ll} \min_{\tau_i} & \rho T_k(\rho) \\ \text{s.t.} & 1 \le \tau_i \le n_i \\ & p \ge p_{\min}^k \Pi_{i=1}^{k-1}\tau_i + \displaystyle\sum_{i=1}^{k-2} m_{i+1}\Pi_{j=1}^{i}\tau_j + m_1 \\ & w_i\tau_i = n_i \\ & w_i \in \mathbf{Z}, \end{array}$$

where $\mathbf{Z}$ is the set of integers. The first constraint reflects the fact that we can reasonably restrict τ_i to $1 \le \tau_i \le n_i$, since otherwise there will always be idle processors.

If we assume that T_k is monotonically decreasing and that A_k exhibits sublinear speedup, then $\rho T_k(\rho)$ is monotonically increasing in ρ. Thus minimizing $\rho T_k(\rho)$ is equivalent to minimizing $-\Pi_{i=1}^{k-1}\tau_i$, so we can solve the equivalent problem

$$\begin{array}{ll} \min_{\tau_i} & -\Pi_{i=1}^{k-1}\tau_i \\ \text{s.t.} & 1 \le \tau_i \le n_i \\ & p \ge p_{\min}^k \Pi_{i=1}^{k-1}\tau_i + \sum_{i=1}^{k-2} m_{i+1}\Pi_{j=1}^{i}\tau_j + m_1 \\ & w_i\tau_i = n_i \\ & w_i \in \mathbf{Z} \end{array} \quad (1)$$

For a given p, there may be no feasible solutions to this problem because of the integrality constraint on τ_i (for example, consider the case when the n_i are large prime numbers). Appendix A analyzes the optimal solutions for this problem when we relax the integrality constraint on τ_i. To illustrate the range of solutions to (1), consider the cases where all m_i values are one or zero. If the m_i values are all one, then a self-scheduling algorithm is used at all levels, and dedicated processors are used for scheduling. In this case, there exists i such that $\tau_1 = \ldots = \tau_{i-1} = 1$, $\tau_i > 1$ and $\tau_j = n_j$ for $j > i$. These solutions maximize the number of simultaneous evaluations of A_k, which effectively minimizes the number of processors that A_k is run on. This intuitively makes sense because the parallel work of A_k is minimized when it is run on $p_{\min}^k$ processors. We expect that integral solutions will be similarly biased towards minimizing the number of processors.

The case where the m_i values are all zero represents a static scheduling policy, which does not use master processes. If the n_i values are sufficiently large, then the optimal solution is when $\tau_1 = \ldots = \tau_{k-1} = \left(p/p_{\min}^k\right)^{1/(k-1)}$. This solution maximizes the product of the τ_i, and it is noteworthy that when integral solutions do exist, they may not be unique. This can happen, for instance, when the n_i share common divisors. If $n_i = 12$ and $n_j = 18$, then a solution with $\tau_i = 6$ and $\tau_j = 9$ is equivalent to a solution with $\tau_i = 3$ and $\tau_j = 18$. Thus concurrency at different levels is interchangable given prudent selections that avoid idleness.

In the general case, an optimal solution to (1) begins with zero or more τ_k at their lower bound and ends with zero or more τ_k at their upper bound. Between these regions, is a set of consecutive indeces $\{i\}$ for which the values m_{i+1} are zero, but within that set the τ_i are either at their upper bounds or at a value that is the same for all other unconstrained τ_k's.

Implementation

In order to maximize concurrency and achieve near-linear scaling, we seek the simultaneous exploitation of many different sources of parallelism by modularizing the parallelism facilities into a reusable software component and then nesting these facilities, one level within the next, through recursive partitioning. MPI [11] provides a convenient mechanism for modularizing parallelism through the use of "communicators." A communicator defines the context of processors over which a message passing communication occurs. By providing mechanisms for subdividing existing communicators into new partitions and for sending messages within and between the new partitions, a lower level of parallelism can be created and managed with a new set of subdivided communicators. And since each new communicator can be further subdivided, multiple levels of parallelism can be nested, one within the other. In this way, massive parallelism can be achieved through the simultaneous exploitation of algorithmic coarse-grained, function evaluation coarse-grained, and function evaluation fine-grained parallelism sources.

The DAKOTA toolkit [12] is a software framework for systems analysis, encompassing optimization, parameter estimation, uncertainty quantification, design of computer experiments, and sensitivity analysis. It interfaces with a variety of simulation codes from a range of engineering disciplines, and it manages the complexities of a broad suite of capabilities through the use of object-oriented abstraction, class hierarchies, and polymorphism [13]. Through implementation in DAKOTA, the impact of investments in parallel code development can be maximized since capabilities developed for optimization also enable parallelism in uncertainty quantification, parameter estimation, and other toolbox capabilities.

Levels of parallelism

While the theory in this paper covers an arbitrary number of parallelism levels, DAKOTA currently supports up to four levels of nested parallelism (which implies three levels of scheduling control). From top to bottom, they are as follows:

Level 1. *Concurrent iterator strategies:* coarse-grained parallelism is realized through the concurrent execution of multiple optimizations/quantifications within a high-level strategy such as parallel branch and bound, optimization under uncertainty, multi-start local search, island-model GAs, etc.

Level 2. *Concurrent function evaluations within each iterator:* coarse-grained parallelism is realized through the concurrent execution of multiple function evaluations within each optimization.

Level 3. *Concurrent analyses within each function evaluation:* coarse-grained parallelism is exploited when multiple separable simulations are performed as part of evaluating the objective function(s) and constraints.

Level 4. *Multiple processors for each analysis:* fine-grained parallelism is exploited when parallel analysis codes are available. For this level, no scheduling by DAKOTA is required as the simulation code is responsible for internally distributing work among the simulation processors.

In terms of the classification of parallelism sources given in the *Introduction*, Levels 1 and 2 are examples of algorithmic coarse-grained parallelism, Level 3 is function evaluation coarse-grained parallelism, Level 4 is function evaluation fine-grained parallelism, and algorithmic fine-grained parallelism is not supported in DAKOTA (the SAND approach is under investigation separately). It is relatively rare for any given application to be able to exploit all four levels; rather each new application selects as many as are applicable from the toolset.

This marks a significant increase in available concurrency from the two-level capability reported in [1] to the four levels currently available. It is important to recognize that the effect on concurrency is not additive, but rather multiplicative in nature. For example, if four concurrent optimizations can be run within a strategy, each optimization having ten independent function evaluations on each cycle, and each function evaluation having three independent analyses, then the fine-grained parallelism available in the analysis can be augmented with 120-fold coarse-grained parallelism.

Partitioning of levels

In DAKOTA, each tier within the nested parallelism hierarchy can use either of two processor partitioning models: a "dedicated master" partitioning in which a single processor is dedicated to scheduling operations and the remaining processors are split into server partitions, or a "peer partition" approach in which the loss of a processor to scheduling is avoided. These models are depicted in Figure 2. The peer partition is desirable since it utilizes all processors for computation; however, it requires either the use of sophisticated mechanisms for distributed scheduling or a problem for which static scheduling of concurrent work performs well (see *Scheduling within levels* below). To recursively partition the subcommunicators of Figure 2, `COMM1/2/3` in the dedicated master or peer partition case would be further subdivided using the appropriate partitioning model for the next lower level of parallelism.

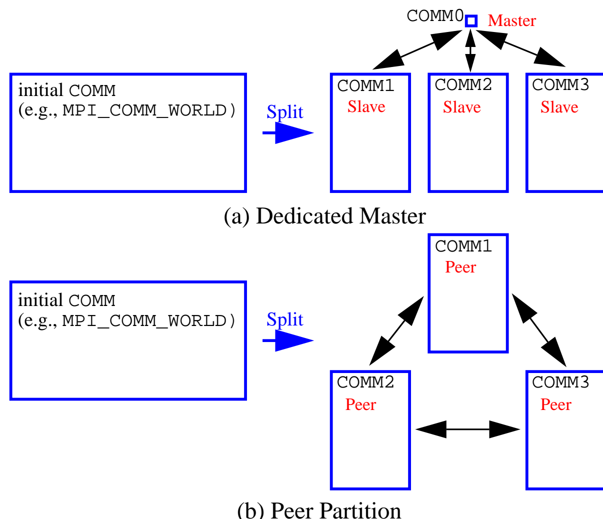


Figure 2. Communicator partitioning models.

Scheduling within levels

Several scheduling approaches are available within Levels 1, 2, and 3:

- *Self-scheduling:* in the dedicated master model, the master processor manages a single processing queue and maintains a prescribed number of jobs (usually one) active on each slave. Once a slave server has completed a job and returned its results, the master assigns the next job to this slave. Thus, the slaves themselves determine the schedule through their job completion speed. Heterogeneous processor speeds and/or job lengths are naturally handled, provided there are sufficient instances scheduled through the servers to balance the variation.
- *Static scheduling:* if scheduling is statically determined at start-up, then no master processor is needed to direct traffic and a peer partitioning approach is applicable. If the static schedule is a good one (ideal conditions), then this approach will have superior performance. However, heterogeneity, when not known *a priori*, can very quickly degrade performance since there is no mechanism to adapt.
- *Distributed scheduling:* in this approach, a peer partition is used and each peer maintains a separate queue of pending jobs. When one peer's queue is smaller than the other queues, it requests work from its peers (hopefully prior to idleness). In this way, it can adapt to heterogeneous conditions, provided there are sufficient instances to balance the variation. Each partition performs communication between computations, so no processors are dedicated to scheduling. Furthermore, it distributes scheduling load beyond a single processor, which can be important for large numbers of concurrent jobs (whose scheduling might overload a single master) or for fault tolerance (avoiding a single point of failure). However, it involves relatively complicated logic and additional communication for queue status and job migration, and its performance is not always superior since a partition can become work-starved if its peers are locked in computation (Note: this logic can be somewhat simplified if a separate thread can be created for communication and migration of jobs).

DAKOTA is designed to allow the freedom to configure each parallelism level with either dedicated master/self-scheduling, peer partition/static scheduling, or peer partition/distributed scheduling. For example, Figure 3 shows the common case in which a concurrent-optimizer strategy employs peer partition/distributed scheduling at level 1, each optimizer partition employs concurrent function evaluations in a dedicated master/self-scheduling model at level 2, and each function evaluation partition employs concurrent multiprocessor analyses in a peer partition/static scheduling model at level 3. In this case, MPI_COMM_WORLD is subdivided into optCOMM1/2/3/.../τ_1, each optCOMM is further subdivided into evalCOMM0 (master) and evalCOMM1/2/3/.../τ_2 (slaves), and each slave evalCOMM is further subdivided into analCOMM1/2/3/.../τ_3.

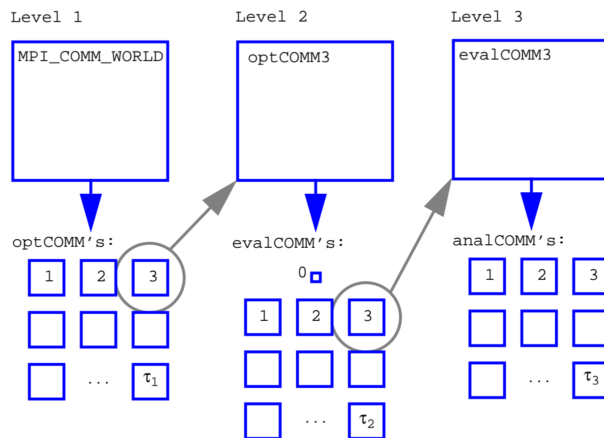


Figure 3. Recursive partitioning for nested parallelism.

Experiments

Two types of computational experiments are of interest:

Investigation of usage models: The mechanisms used to manage concurrent multiprocessor simulations on MP computers are of high importance. These studies compare nonintrusive approaches (asynch, master-slave, and hybrid) with the goal of approximating the performance of the direct, intrusive approach.

Investigation of partitioning schemes: The τ_i recom-

mendations from the mathematical analysis will be demonstrated by fixing the total processor allocation (p), varying τ_i within p, and measuring relative performance.

Usage model computational experiments

Usage model studies are being explored on Sandia's computational plant (Cplant^{TM}) architecture [14], a tightly-connected cluster of DEC alphas running LINUX [15]. This computer, as well as all other Sandia MP computers, uses a service/compute node distinction. Service nodes run full UNIX and are used for job initiation and management. Compute nodes run a minimal operating system (no system calls, forks, or multithreading) whose small footprint leaves as much memory as possible for the application. In this environment, the most computationally efficient way to perform parallel optimization using simulation codes in a nested analysis and design (NAND) approach is to perform a "direct" interface which links the simulation into the optimization software. The combined executable is then executed on the compute nodes where it employs MPI communicator partitions to manage embedded multiprocessor simulations. Unfortunately, this approach is often impractical due to the required intrusiveness into the simulation code (modification to a callable subroutine and MPI communicator modularity required). For this reason, these studies are benchmarking nonintrusive interfacing techniques (no simulation modifications required) which have the goal of approximating the performance of the direct interface. This involves the combined usage of service nodes, where the parallel optimization executes, and compute nodes, where the parallel simulations execute.

Three nonintrusive approaches have been explored on Cplant (Figure 4). In all cases, DAKOTA runs on one or more service nodes and launches simulations on the compute nodes using a job initiation utility. The first approach, called "asynch," involves the execution of DAKOTA on a single service-node processor. DAKOTA uses its asynchronous job initiation (e.g., background system call, nonblocking fork) to manage multiple concurrent jobs. The second approach, called "master-slave," involves the execution of DAKOTA in parallel across multiple service-node processors. Each slave service node uses a synchronous job initiation method (foreground system call, blocking fork) to manage a single job per service node. The third approach, called "hybrid," combines the first two approaches in managing multiple asynchronous jobs on each slave service node. This latter approach was motivated by the observation that spreading application load across service nodes is beneficial, but the concurrency achievable in

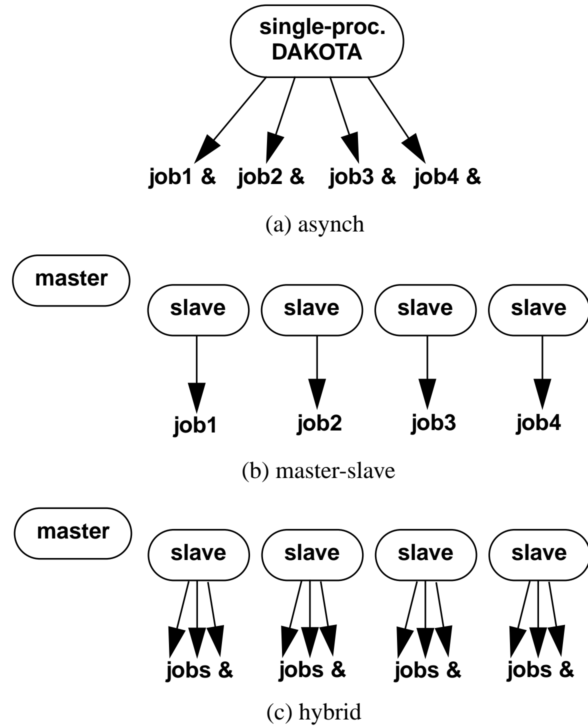


Figure 4. Nonintrusive DAKOTA options.

the master-slave approach is limited by the (relatively small) number of service nodes. The hybrid approach spreads the application load and still allows significant capacity beyond the number of service nodes.

In a design application employing the ALEGRA simulation code, Figure 5 shows the timing for the asynch, master-slave, and hybrid approaches for increasing number of concurrent ALEGRA simulations. Each simulation writes data to a disk local to the service node launching the simulation. Ideally, these curves would all have zero slope since the simulations are the same size and run concurrently. However, competition for service node resources has a detrimental effect, and

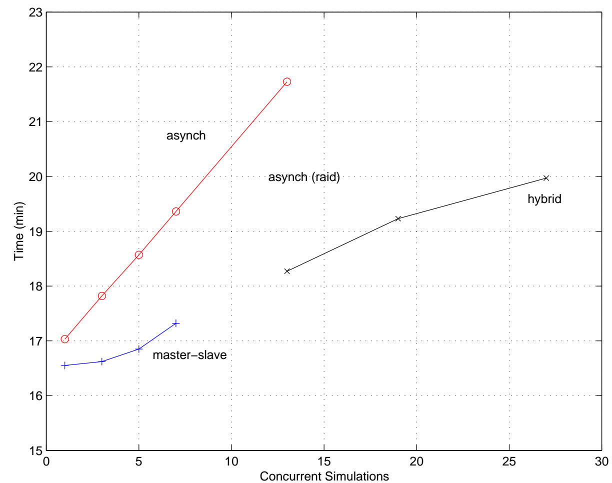
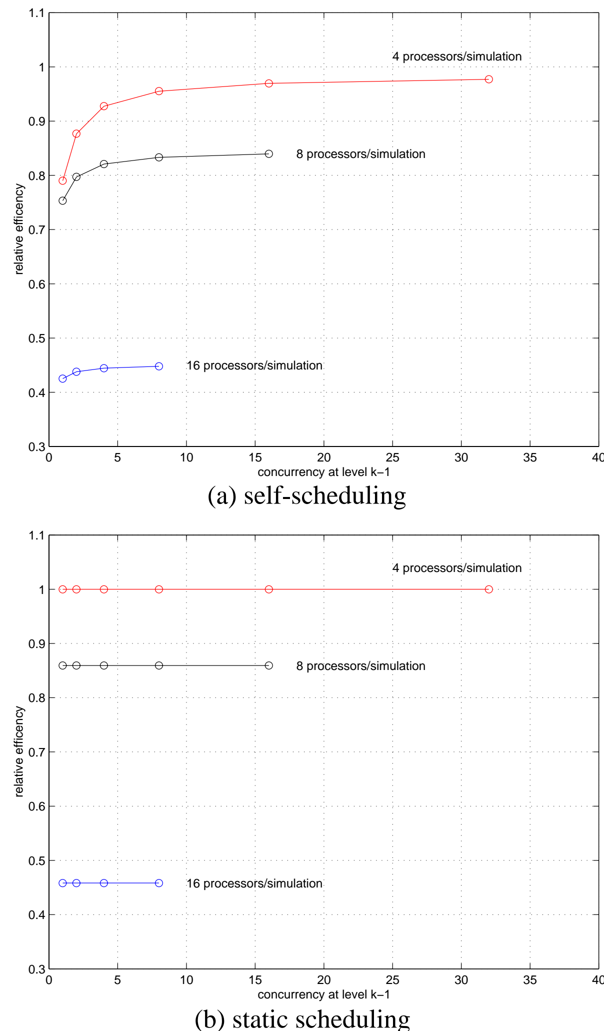


Figure 5. Cplant execution times for serial ALEGRA

spreading the load across multiple service nodes is clearly beneficial. It is evident that the hybrid method provides a natural continuation of the master-slave approach and provides superior performance for larger numbers of concurrent jobs.

Partitioning scheme computational experiments

Given a high performing usage model, the partitioning schemes recommended by the mathematical analysis can be explored. Since performing large-scale verifications is extremely expensive and must have strong mission ties to be justified, a run time simulator has been developed to compare the numerous τ_i configurations that are possible when four levels of parallelism are considered. The simulator is based on the equation for $T(p)$. Wall clock time required for a multiprocessor analysis code ($T_k(p'_k)$) is modeled using experimental data from a series of MP structural dynamics simulations, and since this time dominates in practice, T_i^{serial} and T_i^{comm} are taken to be zero. Figure 6 shows efficiency results from the simulator for fixing the total number of processors at 128 plus masters (including a varying number of masters within a fixed total would cause heterogeneous partitions within a level and would not be done in practice), fixing maximum concurrencies at n_1=1 and n_2=n_3=32, setting p'_k to 4, 8, and 16 processors per simulation (corresponding to 32, 16, and 8 total concurrency in simulations, respectively), and varying the lowest level concurrency τ_3. In all cases, any of the total concurrency not present in τ_3 appears in τ_2 such that the product is constant for each curve. It is evident for the self-scheduling case (Figure 6(a)) that highest efficiencies are achieved for minimum simulation size (4 processors) and for maximized concurrency at the lowest level (32 instances), since this equates to fewer processors dedicated to scheduling (3 masters for the $\tau_1/\tau_2/\tau_3$ = 1/1/32 configuration versus 34 masters for the 1/32/1 configuration). For the static scheduling case (Figure 6(b)), highest efficiencies are again achieved for minimum simulation size, but the concurrencies between τ_2 and τ_3 are interchangeable. Thus, the recommendations from the mathematical analysis hold exactly in the deterministic simulation duration case.

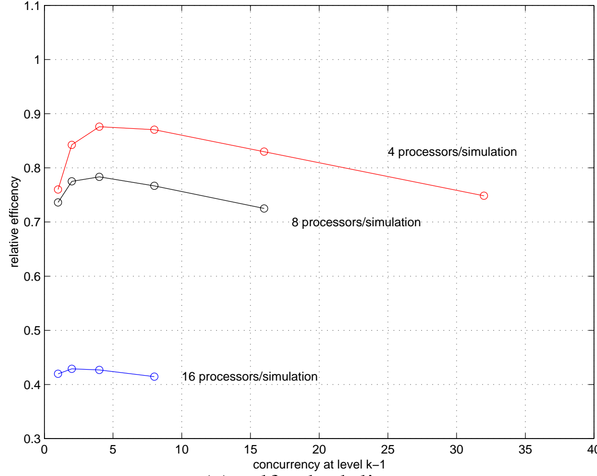


(a) self-scheduling

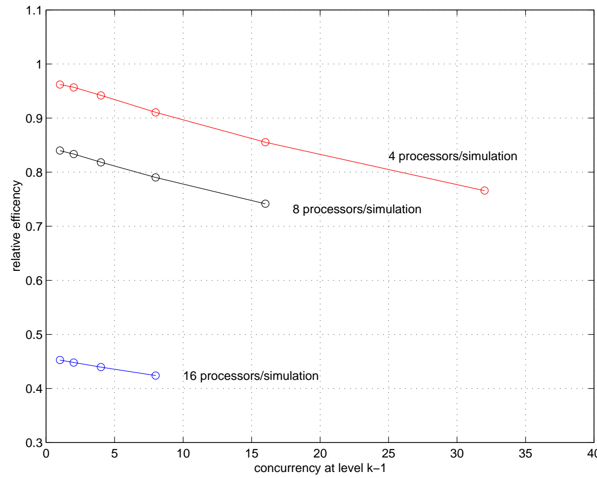
(b) static scheduling

Figure 6. Deterministic simulation duration

However, Figure 7 shows a different trend for a stochastic case in which a 10% variation is added to the simulation duration using an exponential distribution. This is motivated by the fact that some applications can have considerable variability in simulation duration, particularly when the event of interest is dependent on the design variables (e.g., see heat transfer and nonlinear mechanics applications in [16]). For each configuration, 1000 experiments are run with different random seeds and the results are averaged. It is evident that variation in simulation duration is most detrimental to efficiency at the higher τ_3 concurrencies, which is intuitive since this equates to fewer passes through the servers (lower γ_3) and fewer opportunities to balance heterogeneity in job length. Thus, a trade-off between minimizing masters and the need to balance heterogeneity is created in the self-scheduling case (Figure 7(a)) and efficiency is actually maximized in the interior of the curves. Additional experimentation has found that the slope in the large τ_3 region transitions between positive slope (e.g., Figure 6(a)) and negative slope (e.g., Figure 7(a)) at 1% variability in simulation duration. This can be considered to be the boundary of validity of the assumptions in the theoretical analysis for this problem. In the static scheduling case (Figure 7(b)), concurrencies are no longer interchangeable and



(a) self-scheduling



(b) static scheduling

Figure 7. Stochastic simulation duration

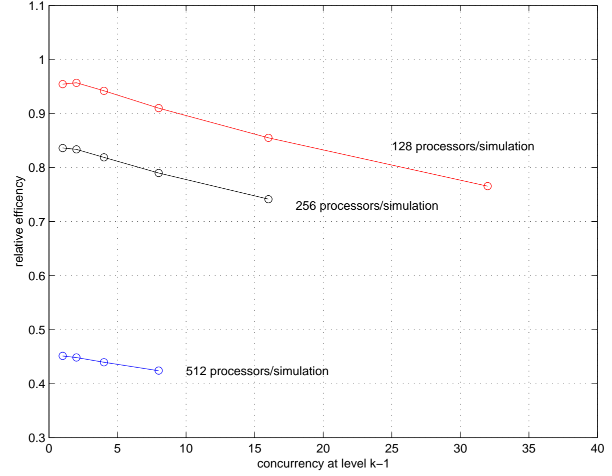


Figure 8. Large-scale, self-scheduling, stochastic duration

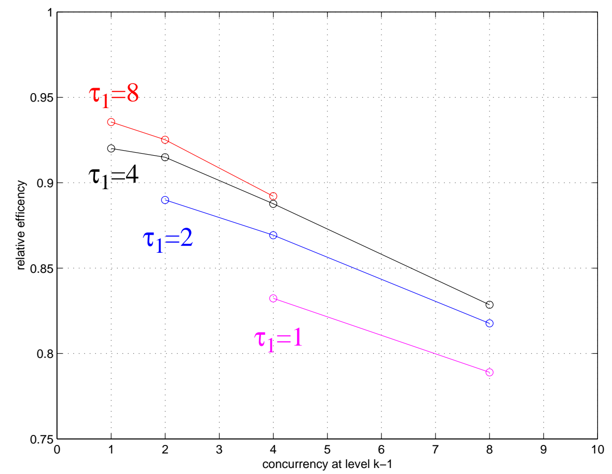


Figure 9. Large-scale, self-scheduling, stochastic duration for fixed values of τ_1

maximizing γ_3 instead of τ_3 is preferred. Presumably, with any variability in simulation duration, this static scheduling conclusion would hold.

Figure 8 shows a large-scale case with an identical set-up to Figure 7(a) except that 128, 256, and 512 processors per simulation are used instead of 4, 8, and 16. This corresponds to a total of 4096 processors plus masters. In this case, the effect of a variable number of masters can be seen to be much lower relative to Figure 7(a). That is, large simulation sizes can mask variations in the number of masters, making this concern secondary to the concern of balancing heterogeneity.

Figure 9 shows another large-scale, self-scheduling, stochastic duration case in which the size of p'_k is fixed at 128 processors per simulation, p is fixed at 4096 (32 total simulation concurrencies) plus masters, and maximum concurrencies are fixed at n_1=n_2=n_3=8. All combinations of $\tau_1/\tau_2/\tau_3$ resulting in 32 concurrencies are shown with curves drawn for fixed values of τ_1. It is evident that maximum efficiency occurs for $\tau_1/\tau_2/\tau_3$ = 8/4/1, or maximized concurrencies at the *higher* levels (the exact opposite of the theory prediction). This configuration results in less replication of the idleness caused by heterogeneity.

Conclusions

Parallel optimization on large numbers of processors has been investigated. Nested parallelism is employed to maximize the number of coarse-grained instances, which allows the execution of fine-grained parallel simulations on minimal partitions where computation could far predominate over communication. This preference of coarse-grained over fine-grained has been consistently verified in computational experiments.

A more subtle point is preference among different sources of coarse-grained parallelism. Theoretical analysis has shown the optimal partitioning schemes in mul-

tilevel optimization algorithms for idealized self-scheduling and static scheduling cases. In the self-scheduling case, it is recommended to give preference to the lower level concurrencies since this minimizes the number of processors lost to scheduling operations. In the static scheduling case, it is shown that little difference exists between the scheduling levels and that their concurrencies may be interchangeable. These recommendations were verified in computational experiments for the case where simulation duration is fixed. With variability in simulation duration, however, the simplifying assumptions in the analysis become violated, and the need to balance heterogeneity (whether due to simulation length, processor speed, or other factors) can become more of a dominant concern than minimizing the number of scheduling processors. In this case, the opposite recommendations hold, that is, maximizing concurrencies at the higher levels.

Future extensions to the mathematical analysis will focus on the issue of simulation duration heterogeneity, since this has been shown to be a dominant concern for large scale applications. Future DAKOTA development will seek to minimize efficiency losses due to replicated scheduling processors by emphasizing static and distributed scheduling approaches over self-scheduling approaches, particularly at the lower parallelism levels where dedicated scheduling processors are most replicated. This will allow DAKOTA partitioning logic to fully endorse maximized concurrency at the higher levels and lead to high performing configurations which are robust in the presence of heterogeneity.

References

[1]Eldred, M.S. and Hart, W.E., "Design And Implementation Of Multilevel Parallel Optimization On The Intel Teraflops," paper AIAA-98-4707 in *Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization (MA&O)*, St. Louis, MO, Sept. 2-4, 1998, pp. 44-54.

[2]Kroo, I., "MDO for Large-Scale Design," *Multidisciplinary Design Optimization State of the Art*, Alexandrov, N.M., and Hussaini, M.Y., eds., Society for Industrial and Applied Mathematics, Philadelphia, 1997, pp. 22-44.

[3]Orozco, C.E., and Ghattas, O.N., "A Reduced SAND Method for Optimal Design of Non-linear Structures," *International Journal for Numerical Methods in Engineering*, Volume 40, 1997, pp. 2759-2774.

[4]Dennis, J.E., and Lewis, R.M., "Problem Formulations and Other Optimization Issues in Multidisciplinary Optimization," AIAA Paper 94-2196, *AIAA Symposium on Fluid Dynamics*, Colorado Springs, CO, June 1994.

[5]Schnabel, R.B., "A View of the Limitations, Opportunities, and Challenges in Parallel Nonlinear Optimization," *Parallel Computing*, Volume 21, 1995, pp. 875-905.

[6]Biedron, R.T., Mehrotra, P., Nelson, M.L., Preston, F.S., Rehder, J.J., Rogers, J.L., Rudy, D.H., Sobieski, J., and Storaasli, O.O., "Compute as Fast as the Engineers Can Think!," NASA/TM-1999-209715, September 1999.

[7]Eldred, M.S., Hart, W.E., Bohnhoff, W.J., Romero, V.J., Hutchinson, S.A., and Salinger, A.G., "Utilizing Object-Oriented Design to Build Advanced Optimization Strategies with Generic Implementation," paper AIAA-96-4164 in *Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on MA&O*, Bellevue, WA, Sept. 4-6, 1996, pp. 1568-82.

[8]Hutchinson, S.A., Shadid, J.N., Moffat, H.K., and Ng, K.T., "A Two-Level Parallel Direct Search Implementation for Arbitrarily Sized Objective Functions," *Proceedings of the Colorado Conference on Iterative Methods*, Breckenridge, Colorado, 1994.

[9]Guruswamy, G.P., "User's Manual for HiMAP: A Portable Super Modular 3-Level Parallel Multidisciplinary Analysis Process," NASA/TM-1999-209578, September 1999.

[10]Eldred, M.S., and Schimel, B.D., "Extended Parallelism Models For Optimization On Massively Parallel Computers," paper 16-POM-2 in *Proceedings of the 3rd World Congress of Structural and Multidisciplinary Optimization*, Buffalo, NY, May 17-21, 1999.

[11]Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J., *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996.

[12]Eldred, M.S, Bohnhoff, W.J., and Hart, W.E., "DAKOTA, An Object-Oriented Framework for Design Optimization, Parameter Estimation, Sensitivity Analysis, and Uncertainty Quantification," Sandia Technical Report SAND00-XXXX, In preparation. Draft available from http://endo.sandia.gov/DAKOTA/papers/Dakota_online.pdf.

[13]Stroustrup, B., *The C++ Programming Language*, 2nd ed., Addison-Wesley, New York, 1991.

[14]Eldred, M.S., Schimel, B.D., van Bloemen Waanders, B.G., and Giunta, A.A., "Optimization with DAKOTA on Cplant," internal report, January 2000.

[15]http://www.cs.sandia.gov/cplant/

[16]Eldred, M.S., Outka, D.E., Bohnhoff, W.J., Witkowski, W.R., Romero, V.J., Ponslet, E.R., and Chen, K.S., "Optimization of Complex Mechanics Simulations with Object-Oriented Software Design," *Computer Modeling and Simulation in Engineering*, Vol. 1, No. 3, August 1996.

Appendix A

This appendix describes the optimal solutions to problem (1). Consider the following reformulation of problem (1), which puts all constraints in a common form:

$$\begin{array}{ll} \min_{\tau_i} & -\Pi_{i=1}^{k-1}\tau_i \\ \text{s.t.} & \tau_i - 1 \geq 0 \\ & -\tau_i + n_i \geq 0 \\ & p - p_{\min}^k \Pi_{i=1}^{k-1}\tau_i \\ & \quad - \sum_{i=1}^{k-2} m_{i+1}\Pi_{j=1}^{i}\tau_j - m_1 \geq 0. \end{array} \quad (2)$$

The Lagrangian for this problem is

$$L(\overline{\tau}, \overline{\lambda}) = -\Pi_{i=1}^{k-1}\tau_i - \sum_{i=1}^{2k-1} \lambda_i c_i(\overline{\tau}),$$

where c_i refers to the i-th constraint in Equation (2). A necessary condition for the optimality of a point $\overline{\tau}^*$ is that there exists $\overline{\lambda}^*$ such that

$$\frac{\partial L}{\partial \tau_i}(\overline{\tau}^*, \overline{\lambda}^*) = 0$$

and $\lambda_i^* \geq 0$ for $i = 1, \ldots, 2k-1$ (e.g. see Gill, Murray and Wright [11]). The following lemma proves the basic fact needed to characterize $\overline{\tau}^*$.

Lemma 1 *Let $1 \leq r < s \leq k-1$, and suppose that $\tau_r^* > 1$. If there exists $m_i = 1$ for $i \in \{r+1, \ldots, s\}$, then $\tau_s^* = n_s$. Otherwise if $\tau_s^* \neq \tau_r^*$ then either $\tau_r^* = n_r < \tau_s^*$, or $\tau_r^* > \tau_s^* = n_s$.*

Proof. Note that the λ_i^* have the property that if constraint i is not tight then $\lambda_i^* = 0$. Thus if $\tau_r^* > 1$ then $\lambda_r^* = 0$ and $\lambda_{r+k-1}^* \geq 0$. Consider the case where there exists $m_i = 1$ for $i \in \{r+1, \ldots, s\}$, and assume towards a contradiction that $\tau_s^* < n_s$. This implies that $\lambda_s^* \geq 0$ and $\lambda_{s+k-1}^* = 0$. Because of the optimality of $\overline{\tau}^*$, we have

$$\begin{aligned} \tfrac{\partial L}{\partial \tau_r}(\overline{\tau}^*, \overline{\lambda}^*) &= -\tfrac{\Pi_{i=1}^{k-1}\tau_i^*}{\tau_r^*} - \lambda_r^* + \lambda_{r+k-1}^* \\ &\quad + \lambda_{2k-1}^* \left(\tfrac{p_{\min}\Pi_{i=1}^{k-1}\tau_i^*}{\tau_r^*} + \tfrac{\sum_{i=r}^{k-2} m_{i+1}\Pi_{j=1}^{i}\tau_j^*}{\tau_r^*} \right) = 0 \end{aligned}$$

and

$$\begin{aligned} \tfrac{\partial L}{\partial \tau_s}(\overline{\tau}^*, \overline{\lambda}^*) &= -\tfrac{\Pi_{i=1}^{k-1}\tau_i^*}{\tau_s^*} - \lambda_s^* + \lambda_{s+k-1}^* \\ &\quad + \lambda_{2k-1}^* \left(\tfrac{p_{\min}\Pi_{i=1}^{k-1}\tau_i^*}{\tau_s^*} + \tfrac{\sum_{i=s}^{k-2} m_{i+1}\Pi_{j=1}^{i}\tau_j^*}{\tau_s^*} \right) = 0. \end{aligned}$$

Now $\lambda_r^* = 0$ and $\lambda_{s+k-1}^* = 0$, so we can rewrite these equations as

$$\begin{aligned} -a + \lambda_{r+k-1}^*\tau_r^* + \lambda_{2k-1}^* b &= 0 \quad (3) \\ -a - \lambda_s^*\tau_s^* + \lambda_{2k-1}^* c &= 0, \quad (4) \end{aligned}$$

where $a > 0$ and $b > c > 0$ since we have $m_i = 1$ for some $i \in \{r+1, \ldots, s\}$. Now from Equation (4) we can conclude that $\lambda_{2k-1}^* > 0$, since the first term is negative and the second term is nonpositive. Combining Equations (3) and (4), we get

$$\lambda_{r+k-1}^*\tau_r^* + \lambda_s^*\tau_s^* + \lambda_{2k-1}^*(b - c) = 0.$$

Since $b > c$, all of the terms in this equation are positive. Thus this can only be satisified if $\lambda_{r+k-1}^* = \lambda_s^* = \lambda_{2k-1}^* = 0$. But this gives a contradiction since $\lambda_{2k-1}^* > 0$, so we can conclude that $\tau_s^* = n_s$.

Consider the case where there exists $m_i = 0$ for $i \in \{r+1, \ldots, s\}$, and suppose that $\tau_r^* < \tau_s^*$. Let $\tau_r^* = \sigma\tau_s^*$. Since τ_r and τ_s are always multiplied together in the last constraint, this constraint becomes less tight as σ approaches one. Further, the objective is minimized as σ approaches one. It follows that σ must be bounded from above by the upper bound on τ_r^*, so this bound is tight. The same argument applies when $\tau_r^* > \tau_s^*$. ■