DESIGN AND IMPLEMENTATION OF MULTILEVEL PARALLEL OPTIMIZATION ON THE INTEL TERAFLOPS

M.S. Eldred* and W.E. Hart†

Sandia National Laboratories‡
Albuquerque, NM 87185

Abstract

Single-level parallel optimization approaches, those in which either the simulation code executes in parallel or the optimization algorithm invokes multiple simultaneous single-processor analyses, have been investigated previously and been shown to be effective in reducing the time required to compute optimal solutions. However, these approaches have clear performance limitations which point to the need for multiple levels of parallelism in order to achieve peak parallel performance. Managing multiple simultaneous instances of massively parallel simulations is a challenging software undertaking, especially if the implementation is to be flexible, extensible, and general-purpose. This paper focuses on the design for multilevel parallelism as implemented within the DAKOTA iterator toolkit. Various parallel programming models are discussed, although emphasis is given to a master-slave implementation using the Message Passing Interface (MPI). A mathematical analysis is given on achieving peak efficiency in multilevel parallelism by selecting the most effective processor partitioning schemes. This analysis is verified in some computational experiments.

Introduction

Computational methods developed in fluid mechanics, structural dynamics, heat transfer, nonlinear mechanics, and numerous other fields of engineering can be an enormous aid to understanding the complex physical systems they simulate. Often, it is desired to use these simulations as virtual prototypes to obtain an acceptable or optimized design for a particular system. This enhances the utility of these computational methods by enabling their use for more than just point solutions; simulation tools can be coupled with optimization methods to automatically determine system performance improvements throughout the product life cycle.

Toward these ends, a general purpose iterator toolkit has been developed for the integration of commercial and in-house analysis capabilities with broad classes of systems analysis tools. Written in C++, the DAKOTA (Design Analysis Kit for OpTimizAtion) toolkit[1] is a flexible, extensible interface between analysis codes and iteration methods. In addition to optimization methods and strategies, the DAKOTA toolkit implements uncertainty quantification with direct and sampling methods, parameter estimation with nonlinear least squares solution methods, and sensitivity analysis with general-purpose parameter study capabilities. By employing object-oriented design to implement abstractions of the key concepts involved in iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for current and future problems of interest. Through DAKOTA, point solutions from simulation codes can be used for answering more fundamental engineering questions, such as “what is the best design?”, “how safe is it?”, or “how much confidence do I have in my answer?”.

In addition to its role as a problem-solving environment, the DAKOTA toolkit also provides a platform for research and development of advanced methodologies which focus on increasing the robustness and efficiency of systems analyses for computationally complex engineering problems. The thrusts of this research are currently (1) the development of sophisticated and adaptive “meta-level” strategies such as multilevel hybrid optimization, sequential approximate optimization, optimization under uncertainty, and parallel branch and bound using algorithm libraries like DOT[2], NPSOL[3], OPT++[4], and SGOPT[5] as building blocks (Figure 1), and (2) the

*Principal Member of Technical Staff, Structural Dynamics Dept., Mail Stop 0439, AIAA senior member.

†Senior Member of Technical Staff, Applied Mathematics Dept., Mail Stop 1110.

‡P.O. Box 5800, Albuquerque, NM 87185, USA.

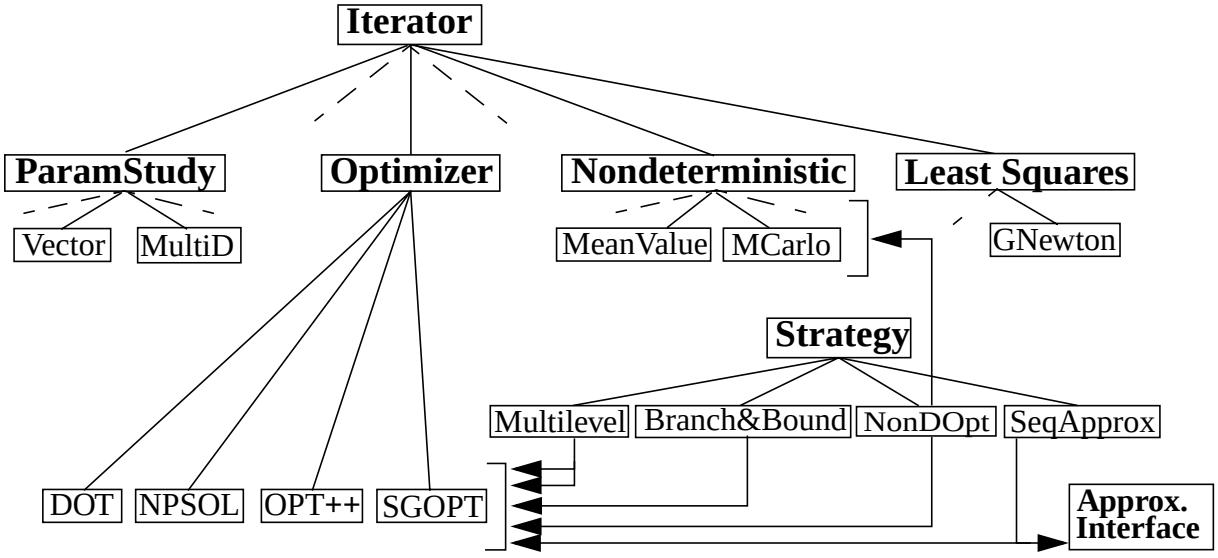


Figure 1. Iterator and strategy class hierarchies within DAKOTA.

development of parallel processing approaches for massively parallel (MP) and distributed architectures, the focus of this paper.

The opportunities for exploiting parallelism in optimization can be categorized into four main areas:

1. *Algorithmic coarse-grained parallelism*: This parallelism involves the exploitation of multiple independent function evaluations. Examples of optimization algorithms containing coarse-grained parallelism include:
   a.) *Gradient-based algorithms:* finite difference gradient evaluations, speculative optimization[6], parallel line search, multiple-secant BFGS[7].
   b.) *Nongradient-based algorithms:* genetic algorithms (GA's), coordinate pattern search (CPS), parallel direct search (PDS)[8], Monte Carlo.
   c.) *Approximate methods:* design and analysis of computer experiments (DACE) evaluations for building response surfaces and training neural networks.
   d.) *Multi-method strategies:* optimization under uncertainty[1], branch and bound[9], multi-start local search[5], island-model GA's[5], GA's with periodic local search[5].
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 simultaneous computation of separable parts of a single function evaluation, where a function evaluation may contain multiple response functions requiring multiple simulations. Examples include separate simulations for multiple objectives and constraint functions, multiple disciplinary analyses for MDO, etc.
4. *Function evaluation fine-grained parallelism:* This involves parallelization of the solution steps within a single analysis code. Examples of Sandia-developed MP analysis codes include PRONTO3D[10], COYOTE[11], MPSalsa[12], ALEGRA[13] PCTH[14], SIERRA[15], etc.

In both the algorithmic and function evaluation cases, coarse-grained parallelization requires very little interprocessor communication and is therefore essentially "free," meaning that there is little loss in parallel efficiency due to communication as the number of processors increases (assuming that there are enough separable computations to utilize the additional processors). 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. The question arises, then, if multiple types of parallelism can be exploited, how should the amount of parallelism at each level be selected so as to maximize the parallel efficiency of the study?

We begin by discussing our prior work with parallel optimization approaches that utilize a single level of parallelism (either parallel computation of independent simulations or parallel computation within a single

simulation - categories 1 or 4 above). We then motivate the need for the application of tools that exploit parallelism at multiple levels and provide a mathematical analysis of multilevel parallelism which seeks to answer the question of how to maximize parallel efficiency. Following a description of parallel programming models, DAKOTA's implementation of a master-slave paradigm using the Message Passing Interface (MPI) standard[16,17] will be presented along with the results from some computational experiments. This implementation is focused on both networks of workstations and the Intel TeraFLOPS supercomputer[18]. The TeraFlops computer, also known as ASCI Red, contains over 9000 Intel Pentium Pro processors and is currently the fastest computer in the world with a peak speed of 1.8 trillion floating point operations per second.

Single-level Parallel Investigations

Single-level approaches which exploit either algorithmic coarse-grained parallelism or function evaluation fine-grained parallelism have been investigated in previous work[20]. These approaches as well as their observed limitations are summarized in the following paragraphs. The worst case fire application uses a parallel optimization algorithm which exploits algorithmic coarse-grained parallelism by invoking multiple independent simulations of single-processor codes, one per processor. The CVD reactor design study demonstrates function evaluation fine-grained parallelism through sequential optimization with an MP simulation code.

Determination of Worst Case Fire Environments

Parallel coordinate pattern search (CPS) from the SGOPT[5] package was used for improving efficiency in optimization of fire surety simulations. Individual thermal simulations executed on nodes of the IBM SP2 using the native loadleveler software to select lightly loaded nodes, and multiple simulations executed simultaneously.

Figure 2 shows the optimization wall clock histories for serial and parallel CPS. With 3 commercial QTRAN licenses, reductions in wall-clock time of a factor of 3 for the parallel CPS optimization were observed over that of serial CPS. With unlimited QTRAN licenses, a factor of 6 savings would have been achievable. These speedups reflect perfect parallel efficiency as expected for this coarse-grained approach since there is negligible communication overhead. When compared with serial nonlinear programming (NLP), relative savings with parallel CPS were a factor of 10 (3 licenses) or 20 (unlimited licenses) since NLP required use of more

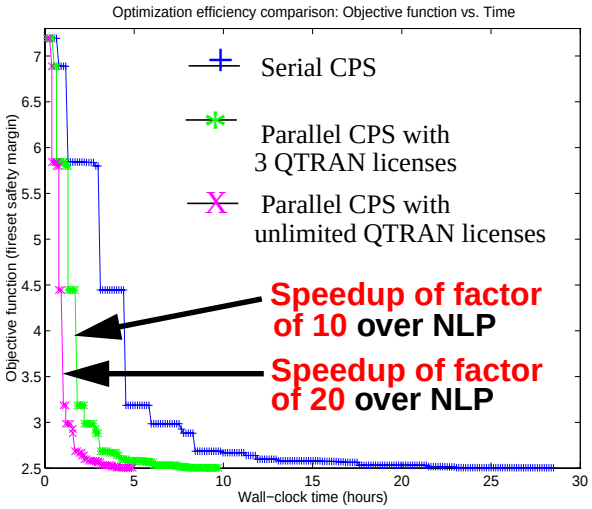


Figure 2. Optimization history comparison: Best objective function value vs. wall-clock time in hours

tightly converged and expensive analyses to enforce smoothness on the order of its finite difference step size.

While pattern search optimizers may use of variety of templates (e.g., PDS[8]), this particular pattern search optimizer executes 2 simulations in each of n parameter directions during an iteration. The end of an iteration is a synchronization point for the parallel algorithm; thus, $2n$ simulations at most may be performed in parallel. Then, the maximum possible parallel speedup relative to serial CPS for the 3 parameter fire surety application using single-processor analyses is 6. This shows clearly the limitation of this single-level approach in that parallel speedup is limited by the number of independent evaluations on a particular optimization cycle. Additional levels of parallelism are needed to extend this performance.

CVD Reactor Design

Massively parallel simulations have been employed in gradient-based optimization studies to allow for expeditious analysis of high fidelity models of the chemically reacting flows within a CVD reactor. MPSalsa simulations[21,22] were executed on a partition of nodes on Sandia's 1840 node Intel Paragon.

A coarse mesh was used for initial optimization studies to efficiently locate promising areas of parameter space. An accurate fine mesh was then used to determine an optimal solution. Figures 3 and 4 show the optimization progression for the coarse and fine meshes, respectively. Each coarse mesh function evaluation took 2-3 minutes on 256 Intel Paragon processors, while each fine mesh function evaluation required 5-8 minutes on 512 processors. For each problem size, there was a trade-off between computational speed-up and interpro-

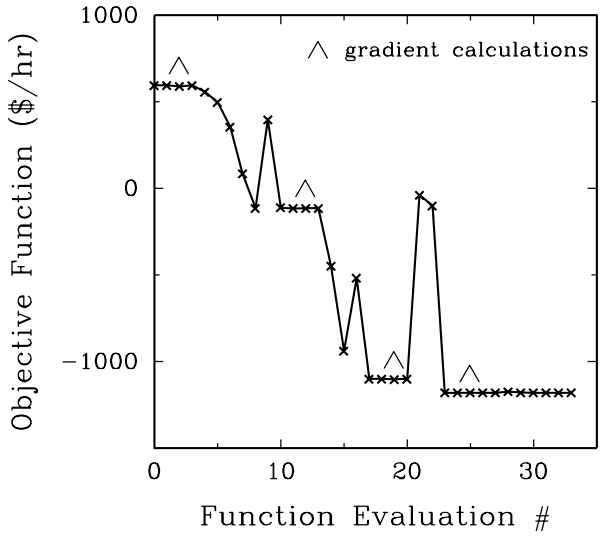


Figure 3. Objective function history for a 3 parameter CVD reactor optimization on a coarse mesh.

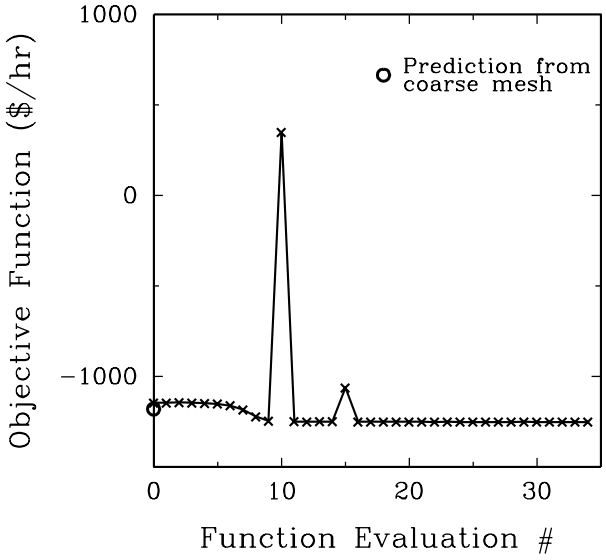


Figure 4. Objective function history for CVD reactor optimization on the fine mesh with initial guess from the coarse mesh converged solution.

cessor communication overhead and these numbers of processors achieved an effective balance for these problem sizes. This points clearly to the efficiency bottleneck in this approach. If, for a given problem size, communication will eventually dominate computation as the number of processors is increased, then the speedup cannot scale with large numbers of processors without introducing additional levels of parallelism.

Multilevel Parallelism

It has been shown that optimization approaches which utilize single-level parallelism can have clear performance barriers. Parallel optimization of single-processor simulations is limited by the number of independent evaluations per cycle, and sequential optimization of parallel analyses is limited by the practical limit on processors that can be used for a single parallel simulation before inter-process communication dominates actual computational work.

These observations point clearly to the need for multilevel parallelism, in which parallel optimization strategies coordinate multiple simultaneous simulations of multiprocessor codes. This approach is not new. Reference [23] shows nearly linear speedup by evaluating the objective functions on the minimum number of processors and by performing several of these in parallel as governed by the independent evaluations of the parallel direct search (PDS) algorithm. For the purposes of DAKOTA, a general implementation beyond the specifics of any one algorithm is needed.

Mathematical Analysis

We motivate our investigation of multi-level parallel optimization methods by analyzing the relative efficiency of a simple, abstract multi-level optimizer. The efficiency of a parallel algorithm is:

$$E(p) = \frac{T(1)}{pT(p)} \tag{1}$$

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. The relative efficiency of a parallel algorithm is

$$\hat{E}(p) = \frac{E(p)}{E(p_{min})} = \frac{p_{min}T(p_{min})}{pT(p)} \tag{2}$$

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

The efficiency of a multilevel optimizer must encompass both the efficiency of the optimizer as well as the efficiency of the parallelized function evaluation. Suppose that each iteration of the optimizer requires κ function evaluations and that each function evaluation uses p' processors. For simplicity, we make the following assumptions:

- The time required to communicate an evaluation request from the master to a slave equals the communication time required to communicate a response from the slave to the master, and these communication times are the same for all communication between master and slaves.
- The time required to execute the function evaluation is constant for all evaluations.
- The overhead imposed on the master for managing the communication requests and responses is negligible.

- That $\kappa \geq \left\lfloor \frac{p-1}{p'} \right\rfloor$ (i.e., the number of evaluations per cycle is ≥ the number of multiprocessor simulation servers).
- That $p' \geq \left\lfloor \frac{p-1}{\kappa} \right\rfloor$ which indicates that when κ is small compared to p - 1, p' is sufficiently large to utilize the allocated processors.

Then the time required for a multilevel master-slave optimizer to optimize a particular problem with p total processors is

$$T(p) = T_{serial} + \alpha\beta_p(2T_{comm} + \bar{T}(p')) \qquad (3)$$

where $\bar{T}(p')$ is the time required to execute a function evaluation on p' processors, T_{comm} is the time required to perform communication between the master and slave processors, T_{serial} is the execution time of the serial portion of the optimization algorithm, α is the number of cycles to convergence, and

$$\beta_p = \left\lceil \frac{\kappa}{\left\lfloor \frac{p-1}{p'} \right\rfloor} \right\rceil \qquad (4)$$

The value β is the maximum number of function evaluations that any slave performs in each iteration of the optimization algorithm. This value provides a worst-case picture of how many function evaluations need to be computed on a server during a cycle, given that $\left\lfloor \frac{p-1}{p'} \right\rfloor$ function evaluations are being computed simultaneously as long as κ contains enough jobs to keep the servers busy and that some servers may be idle towards the end of a cycle. Note that the analysis assumes for simplicity that the remaining $p - 1 - p'\left\lfloor \frac{p-1}{p'} \right\rfloor$ processors are left idle, although in practice, one might take the idle processors and run another simulation (which might lead to improved efficiency overall when κ is large). DAKOTA's design divides the remainder of processors among each of the servers.

Suppose that the minimum number of processors on which a function evaluation can be executed is p'_{min}. Then for a multilevel master-slave optimization algorithm, it follows that

$$\begin{aligned}\hat{E}(p) &= \frac{(p'_{min}+1)T(p'_{min}+1)}{pT(p)} \\ &= \frac{(p'_{min}+1)(T_{serial} + \alpha\kappa(2T_{comm} + \bar{T}(p'_{min})))}{p(T_{serial} + \alpha\beta_p(2T_{comm} + \bar{T}(p')))}\end{aligned} \qquad (5)$$

for $p \geq p'_{min} + 1$. We have used $p_{min} = p'_{min} + 1$ because this is the smallest number of processors on which the master-slave algorithm can run (even though this is a single-level master-slave algorithm with this few processors). A general evaluation of $\hat{E}(p)$ is not possible because of the dependence on α. However, for a given value of α it possible to evaluate the utility of multilevel parallelism in a master-slave design.

When applying a multilevel master-slave algorithm, one generally has a value of κ that is determined by the optimization algorithm, and the number of processors p is selected such that $p \geq p'_{min} + 1$. It remains, then, to determine the value of p' for optimization. At one extreme, $p' = p - 1$, so the algorithm is effectively single-level parallel using a single multiprocessor analysis server. At the other, $p' = max\left\{p'_{min}, \left\lfloor \frac{p-1}{\kappa} \right\rfloor\right\}$, which may also be single-level parallel using multiple single-processor servers if $p'_{min} = 1$ and $\kappa \geq p$ - 1.

Now assume that $\bar{T}(p')$ is a convex, monotonically decreasing function of p'; this assumption is usually satisfied by parallel algorithms. Also, assume that $\bar{T}(p-1)$, the smallest $\bar{T}(p')$, is large relative to T_{comm} and T_{serial}. Since we are particularly interested in simulation-based optimization, this assumption is reasonable. Given these assumptions, we can approximate $\hat{E}(p)$ as

$$\hat{E}(p) \approx \frac{(p'_{min}+1)\kappa\bar{T}(p'_{min})}{p\beta_p\bar{T}(p')} \qquad (6)$$

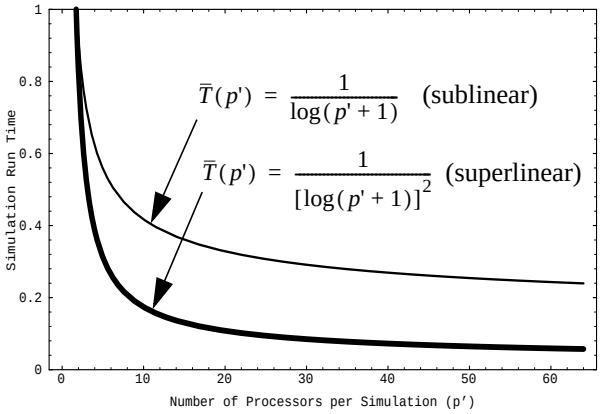
$$= \frac{(p'_{min}+1)\kappa\bar{T}(p'_{min})}{p\left\lceil \frac{\kappa}{\left\lfloor \frac{p-1}{p'} \right\rfloor} \right\rceil \bar{T}(p')} \qquad (7)$$

$$\hat{E}(p) \approx \frac{(p'_{min}+1)(p-1)\bar{T}(p'_{min})}{pp'\bar{T}(p')} \qquad (8)$$

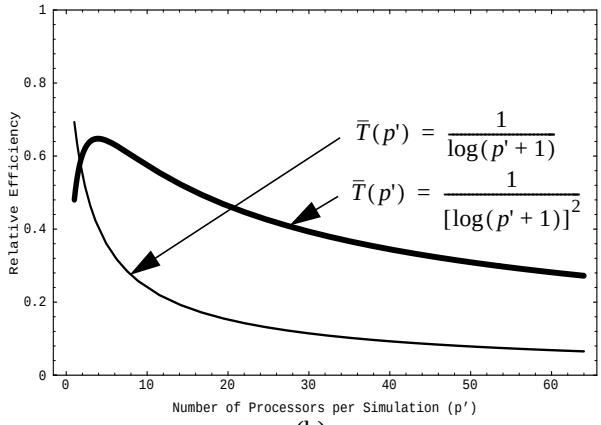
As a function of p', this is a function of the form

$$f(p') = \frac{C}{p'\bar{T}(p')} \qquad (9)$$

for some constant $C > 0$. It follows for $p' \in \{p'_{min}, \ldots, p-1\}$ that $\hat{E}(p)$ is maximized at either $max\left\{p'_{min}, \left\lfloor \frac{p-1}{\kappa} \right\rfloor\right\}$ or at $\frac{df}{dp'} = 0$. Note, however, that f is monotonically decreasing, since $p'\bar{T}(p')$ will be monotonically increasing unless $\bar{T}$ exhibits superlinear speedup. Thus in practice, the maximum value of $\hat{E}(p)$ is at $max\left\{p'_{min}, \left\lfloor \frac{p-1}{\kappa} \right\rfloor\right\}$. Figure 5 illustrates two examples, one with sublinear speedup (generally seen in



(a)



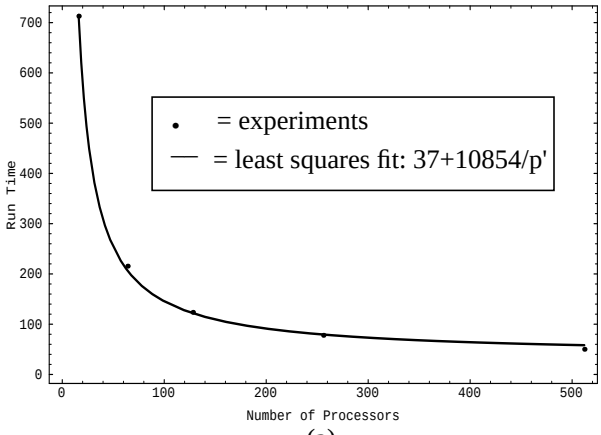
(b)

Figure 5. Illustration of multilevel optimizer performance: (a) run time and (b) relative efficiency for different numbers of processors per simulation.

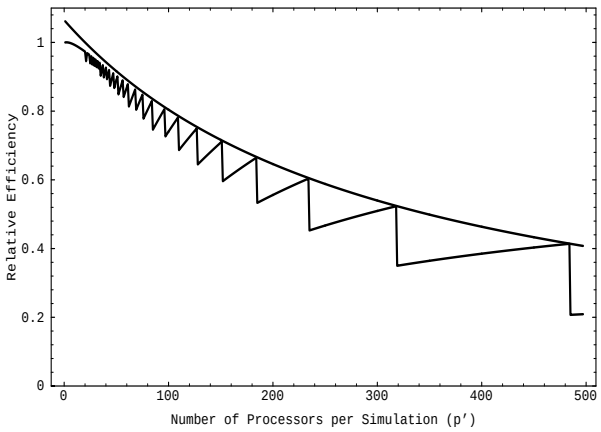
practice, $\hat{E}(p)$ maximized at p'_{min}) and one with superlinear speedup for small p' (rare in practice, $\hat{E}(p)$ maximized at $\frac{df}{dp'} = 0$).

This result demonstrates the utility of multilevel parallelism in a master-slave design. When $p'_{min} > 1$, a multilevel parallel master-slave algorithm is always more efficient than the single-level master-slave algorithm which uses all processors to compute each function evaluation. Furthermore, even if $p'_{min} = 1$, it is common to have $p' > p'_{min}$ because of the lower bound on p' imposed by κ. While the relative efficiency would be higher in this case if p were decreased to $\kappa + 1$, requirements on turn around time will often preclude this possibility. Finally, note that this analysis is not affected by the value of κ except for providing this lower bound on p'.

A final caveat for this analysis is that the approximation in Equation 8 will not be particularly accurate when the floor or ceiling operations round off the fractions in β_p significantly. Note that in a practical context, p would be chosen to be a multiple of p' with one processor added for the master so as to eliminate the processor remainder. Thus we are primarily concerned with the error introduced by the ceiling operation, which involves the previously mentioned issue of idle servers towards the end of κ evaluations in a cycle. Figure 6 illustrates this effect with a run-time curve taken from Shadid and Tuminaro[19]. Figure 6a shows the run-time of the simulation code, along with the curve fitted through these points. Figure 6b shows the curves for Equations 7 and 8 for (p = 1000, p'_{min} = 10, and κ = 1000). Note that the values for Equation 7 are always lower than the curve for Equation 8, and for low values of p', Equation 7 is strictly below Equation 8 because Equation 7 is only



(a)



(b)

Figure 6. Evaluation of round off error: (a) an idealized curve for a numerical simulation and (b) a comparison between the efficiencies of Equation 8 (the top curve) and Equation 7 (the bottom curve), which includes the exact formula for β_p.

evaluated for integral values of p'. However, the discretization error is less pronounced for small p', which have the highest efficiencies. Thus, Equation 8 offers a reasonable approximation.

Underlying Software Design

The DAKOTA (Design Analysis Kit for OpTimizAtion) iterator toolkit[1] utilizes object-oriented design with C++[24] to achieve a flexible, extensible interface between analysis codes and system-level iteration methods. This interface is intended to be very general, encompassing broad classes of numerical methods which have in common the need for repeated execution of simulation codes. DAKOTA's capabilities for mapping parameters into responses are encompassed in an interface abstraction which includes simulation interfacing through system calls or direct function calls, use of approximations such as neural networks and response surfaces, use of internal testing functions, and many other techniques. DAKOTA provides a framework for the implementation of these techniques within the **DakotaInterface** class hierarchy shown in Figure 7, where **SysCall** and **DirectFn** provide mechanisms for interfacing with simulation codes through system calls and direct invocations, respectively, and **ANN**, **RSM**, and **MPA** provide mechanisms for interfacing with artificial neural networks, response surface methods, and multipoint approximations, respectively.

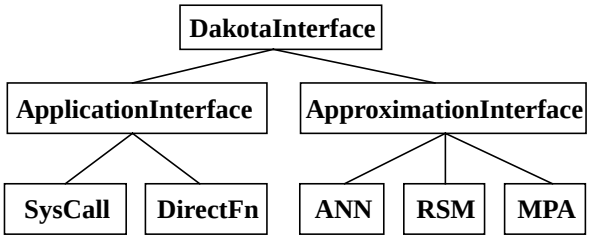


Figure 7. Interface class hierarchy within DAKOTA.

Of particular importance to the parallel optimization discussion are the operations of the **DirectFn** and **SysCall** simulation interfacing classes. Each of these classes has member functions for synchronous and asynchronous parameter to response mappings. For the system call variant, synchronous operation amounts to spawning the system call in the foreground and waiting for its completion, while asynchronous operation involves spawning the system call in the background, continuing with other tasks (e.g., other simulation system calls), periodically checking for process completion, and finally retrieving the results. In the direct function invocation case, synchronous operation involves a standard procedure call to a simulation linked within the code. Asynchronous operation involves the use of multithreading to accomplish multiple simultaneous simulations.

Single-processor DAKOTA. The asynchronous mappings of the **DirectFn** and **SysCall** simulation interfacing classes can be used to accomplish algorithmic coarse-grained parallelism even when the DAKOTA process is running on a single processor. In this case, some additional mechanism external to DAKOTA will usually be desired to distribute the asynchronous jobs among processors, since multitasking on a single processor is generally slower than running the jobs sequentially. For the asynchronous system call case, network load leveling software (as in the worst case fire application on the SP2) or compute server job queues can provide this mechanism, and in the asynchronous direct function case, thread schedulers can be used (e.g., to select nodes within an SMP architecture).

To accomplish multilevel parallelism in this context, one could configure DAKOTA to submit multiple multiprocessor jobs to the queues of a parallel compute server. While swamping the queues with multiple jobs is forbidden in the good citizen rules of the TeraFLOPS machine, it is allowable to allocate a large number of processors to a single script which in turn allocates multiple jobs on partitions of this total processor allocation (this is allowable since the larger jobs do not execute at the same priority as the smaller jobs). This is in fact mimicking the communicator partitioning capabilities of MPI within sophisticated scripting. While this has the advantages of simplifying the automation of pre- and post-processing and minimizing analysis code modifications, it has the strong disadvantage of being highly specific to the job submission software of a particular parallel machine and is therefore not particularly flexible or extensible.

Multiprocessor DAKOTA. When executing DAKOTA in multiprocessor mode using message passing, the synchronous and asynchronous operations of the **DirectFn** and **SysCall** simulation interfacing classes are issues that are local to a processor. Layered on top of these local interfacing capabilities is the software which manages message passing for assignment of work among processors. This design allows flexibility in handling local evaluation mechanisms independently from the particular form of the global message passing model. For example, within the global context of a master-slave approach in which the master is *asynchronously* assigning jobs and retrieving results using message passing with slave servers, it is common for the slave servers to locally execute their simulations using the *synchronous* **DirectFn** or **SysCall** protocols.

For multiprocessor DAKOTA, multilevel parallelism is accomplished by internally managing partitions

of the total processor allocation using MPI communicators. This is described in detail in the "TeraFLOPS Implementation with MPI" section to follow.

Parallel Programming Models

Several parallel programming models have been considered for DAKOTA implementation. Important issues in evaluating these models have been:

1. the need to minimize modifications to analysis codes to avoid problems with commercial software and to avoid creating special-purpose, unsupported versions of in-house codes
2. the need to minimize contention for I/O resources
3. operating system facilities for multitasking, multithreading, and allocating and/or managing computing resources
4. support of heterogeneous executables (MPMD) and/or dependence on standards which are not yet commonly available (e.g., MPI-2[26])

where MPMD denotes "Multiple Program, Multiple Data," which means that heterogeneous executables may be used on different processors using different data. Conversely, SPMD denotes "Single Program, Multiple Data" which is the more common case in which each processor executes the same program, even though the data on each processor may be different.

On MP architectures with lightweight operating systems (e.g., the Cougar OS on the Intel TeraFLOPS[25]), it is common for only one process (and a single thread of execution within the process) to be allowed per processor. That is, there are severe limitations in operating system facilities (issue 3 above) which eliminate the possibilities of system calls and multithreading. In this case, either a.) the analysis code must be modified into a linkable library (issue 1) for use with the synchronous direct function interface, b.) the analysis must be modified to receive messages directly from DAKOTA (issue 1) so that it can be loaded in an MPMD model (issue 4), or c.) the parallel analysis must be created dynamically using facilities from MPI-2 (issue 4). Thus, it can be seen that "lean and mean" MP operating systems, while sufficient for a single application code, can impose significant challenges on optimization software seeking to make use of existing analysis codes. For this reason, a multipurpose parallel design is being implemented within DAKOTA which will run under Cougar restrictions but can exploit the additional capabilities of other environments as well as new OS and message passing capabilities (e.g., MPI-2) as they come on line.

Given these issues, three primary models for parallel optimization have been considered. They are listed in order of increasing sophistication:

1. Peer-Master SPMD - In this model, each processor replicates the basic execution of all of the sequential calculations in DAKOTA. The simultaneous execution of multiple simulations is coordinated in the innermost-loop of the iterator. Under Cougar, the analysis code must be linked directly into the DAKOTA executable which is the same for all processors. This model can have resource contention issues unless care is taken to prevent I/O conflicts among the processors. This model was implemented on the TeraFLOPS as an initial demonstration of a multiprocessor DAKOTA with single-level parallelism; however its contention for I/O led to the implementation of a master-slave model.
2. Master-slave SPMD - In this model, one processor is the master in charge of executing the iterator, while the other processors are slaves responsible only for executing simulations. The master can also execute simulations if the parallelism is single-level algorithmic (although this may introduce a bottleneck if the slaves are waiting for work while the master executes a long simulation), however in the multilevel parallelism case, only the slave servers can execute multiprocessor simulations. The master and slave processors still execute the same program which, in the case of Cougar, must contain both DAKOTA and the analysis code linked together. Unlike the peer-master model, however, they execute different parts of the combined code; the master executes iterator code while the slaves execute function evaluation code. This model eliminates resource contention since the master is responsible for all I/O. Since the slave executable is responsible for little other than coordinating an analysis, it is a small step to an MPMD implementation in which much or all of DAKOTA is removed from the slave executable.
3. Master DAKOTA-slave analysis MPMD - In this case, only one master processor is executing DAKOTA, and the slave processors execute instances of the analysis code. In some cases, the analysis code can be modified to receive messages containing the parameter data and function requests and return messages containing the simulation results. If these modifications are not performed, then DAKOTA will need to utilize data files to communicate with the analysis codes, which can lead to I/O contention on machines with thousands of processors but tens of file devices. There are two variations of this model that reflect how the slave process is initiated. A *server* model would have the slave process running a main loop continuously to service simulation requests from the master. A *dynamic* model would have the master spawn slave processes at run time, using a process-creation utility provided by the operating system or message-passing

library (e.g., MPI-2).

Thus far, the discussion on parallel models has been limited to the case of 2-level parallelism in which a single parallel optimization algorithm invokes multiple instances of multiprocessor simulations. An interesting extension to this is the 3-level parallel master-slave design in which the master executes a strategy and the slaves execute sub-iterators, which may themselves be 2-level parallel. Examples of this type of strategy include optimization under uncertainty, parallel branch and bound, multi-start local search, multiple genetic algorithms with population sharing, and genetic algorithms employing periodic local search.

Parallel Environment Issues

Given the wide range of parallel architectures, it is not surprising that different parallel environments impose different requirements on parallel software. One issue that is particularly relevant to DAKOTA is whether system shells can be used to manage the execution of analysis codes along with their associated pre- and post-processing routines since this is the principle model used to run analysis codes on single workstations. Parallel environments that use the Unix operating system (e.g. the IBM SP2 and networks of workstations) provide system calls that enable analysis codes to be executed through system shells. Unfortunately, this capability is not widely available on MP architectures due to the desire to have an operating system that is small and fast and provides just those basic features needed by a computation[25]. Even the MPI-2 standard[26], which provides a facility for dynamic spawning of executables, will not allow invocation of system shells.

Another basic issue is whether processes can be spawned dynamically. Parallel environments that use the Unix operating system and SMP machines like the SGI Origin 2000 provide the capability to spawn processes from independent executables. This capability may become more widely available on MP machines as implementations of the MPI-2 standard are developed. MPI-2 may also enable MPMD parallel models in which an analysis can be used completely unmodified as an "off-the-shelf" simulation. Unfortunately, the massively parallel computers at Sandia do not yet support MPI-2. On both the Intel Paragon and Intel TeraFLOPS machines, processes are statically allocated to processors when the processors are allocated to the user. However, static allocation does provide for different executables to be allocated together. Consequently, we expect that the Master DAKOTA/Server Slave model will eventually be feasible on these machines (although at this time, statically allocated heterogeneous executables have separate `MPI_COMM_WORLD`s which do not permit intercommunication within a global context).

Finally, it is worth noting that pre- and post-processing in a parallel environment with restricted OS facilities is another significant challenge. If additional tools are required for automated model regeneration and response recovery, then these tools may have to be linked into a combined executable as well. It is hoped that the majority of these operations can be handled with MPI gather, scatter, and reduce operations within the analysis, although this will largely depend on the nature of the parameters being designed and the responses being controlled.

In summary, the currently available MPI-1 implementation on the Intel TeraFLOPS will allow master-slave SPMD multilevel parallelism. It will also permit a basic master DAKOTA-slave analysis MPMD multilevel parallel capability that relies on communication through files (since there is no global context for message-passing intercommunication). We expect that MPI-2 will admit a much improved MPMD environment.

TeraFLOPS Implementation with MPI

The environment provided by the MPI-1 standard is well-suited for the master-slave SPMD model for both single-level and multilevel parallelism. In particular, the global MPI communicator (`MPI_COMM_WORLD`) can directly provide the context needed for single-level parallelism, or for multilevel parallelism, it can be partitioned into new intra-communicators which delineate the set of processors to be used for each multiprocessor analysis. Since these intra-communicators can be passed into a simulation for use as the simulation's computational context, the use of communicators enables the analysis routines to be provided as a generic library utility that can be run on an arbitrary set of processors (which was one of the goals of the MPI standard).

Within DAKOTA, new intra-communicators are created with the `MPI_Comm_split` routine. In order for the master to send messages to the new intra-communicators, inter-communicators are created with calls to `MPI_Intercomm_create`. Once the new communicators are created, the single-level and multilevel algorithms for scheduling jobs from the master are virtually identical (in fact, the single-level case could be handled as a special case of the multilevel case, but the DAKOTA design opted to maintain separate algorithms and avoid the overhead of additional communicators for the single-level case). In addition, communicator partitions can be reallocated multiple times. This enables dynamic repartitioning of `MPI_COMM_WORLD` for each simulation interface within a strategy that manages multiple models (e.g., four 256 processor servers for a coarse model followed by two 512 processor servers for a fine model). This is conveniently managed by allocating a particular communicator partitioning scheme in

the constructor of each interface object.

Implementing the master-slave model within a single executable entails a division of iterator code (master) from function evaluation code (slave). This is accomplished within DAKOTA at the strategy layer[1]. In the strategy constructor, the master processor instantiates the required iterators and models whereas the slave processors instantiate only the required models. When the strategy is executed, the master executes the current iterator and sends analysis requests for the current model to the slaves which run server code bound to the current model. When the master completes iteration on the current model, it sends a termination message to the slaves which then exit the current model. If additional work remains within the strategy, then the process repeats for the next iterator and model. Additional features include: (1) use of a self-scheduling design (also known as a task pool design) to load balance the slave servers in which the first server to return results from the current set of jobs is allocated the next job, (2) the use of buffer packing which allows for send/receive of a heterogeneous set of data within a single message, and (3) use of a ParallelLibrary class hierarchy which encapsulates the specific syntax of message passing operations for particular message passing libraries.

Computational Experiments with DAKOTA

Preliminary timing results are shown in Figure 8 for a cluster of 13 workstations. Minus the master processor, the 12 slave processors can be evenly partitioned into analysis servers containing 2, 3, 4, 6, or 12 processors. The method employed is a parallel parameter study which performs centered one-dimensional parameter studies (11 evaluations each) for each of 50 design variables, giving a total of 550 function evaluations for each run. Each function evaluation involves an internal multiprocessor test simulation which, while not nearly as computationally intensive as an engineering simulation, does reproduce the basic performance issues of interest. It can be seen that total run time is minimized (and efficiency is maximized) for $p'_{min} = 2$ processors per analysis.

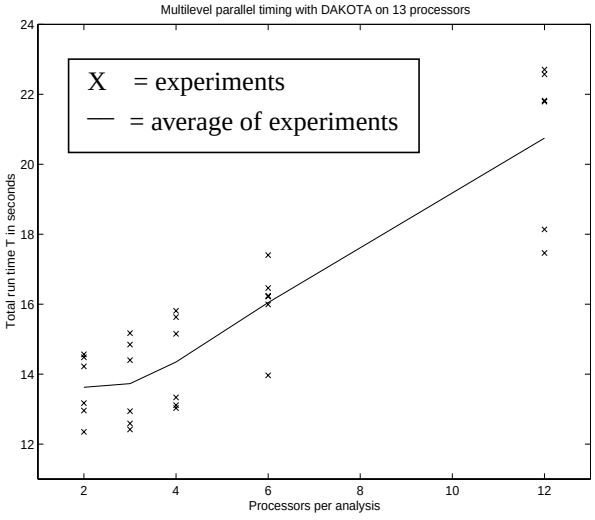


Figure 8. Multilevel parallelism run time for 13 workstations for varying processors per analysis server, p'.

Conclusions

Single-level parallel optimization investigations have been reviewed and both mathematical and computational motivation for multilevel parallelism has been given. Various possible designs for parallel optimization on MP computers have been presented. The DAKOTA software has implemented a self-scheduling master-slave SPMD model using MPI message-passing which performs multilevel parallel optimization on workstation clusters and the Intel TeraFLOPS supercomputer. This implementation will continue to evolve as it seeks to minimize the need for analysis code modifications and to simplify model regeneration and response recovery in the MP environment. These extensions will involve exploiting MPMD capabilities and emerging message passing standards (MPI-2) as they come on line.

In addition, the DAKOTA implementation will be seeking to exploit additional levels of parallelism beyond the two presented. By developing a master-slave design in which the master runs a strategy and slave servers execute sub-iterators (instead of analysis servers), additional algorithmic coarse-grained parallelism can be exploited. Examples of this type of strategy include optimization under uncertainty, parallel branch and bound, multi-start local search, multiple genetic algorithms with population sharing, and genetic algorithms employing periodic local search. This additional level of parallelism can be used to further extend utilization possibilities for very large parallel machines.

Acknowledgments

The authors would like to thank Bill Bohnhoff and Ron Rhea for their assistance with MPI development and Andrew Salinger for his assistance with MPSalsa data used in the mathematical analysis of multilevel parallelism.

References

[1]Eldred, M.S, Bohnhoff, W.J., and Hart, W.E., “DAKOTA, An Object-Oriented Framework for Design Optimization, Parameter Estimation, Sensitivity Analy-

sis, and Uncertainty Quantification," Sandia Technical Report SAND98-XXXX, In preparation. Also available from http://endo.sandia.gov/9234/sd_optim/Dakota.pdf

[2]*DOT Users Manual*, Version 4.10, VMA Engineering, Colorado Springs, CO, 1994.

[3]Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., *User's Guide for NPSOL (Version 4.0): A Fortran Package for Nonlinear Programming*, System Optimization Laboratory, TR SOL-86-2, Stanford University, Stanford, CA, Jan. 1986.

[4]Meza, J.C., "OPT++: An Object-Oriented Class Library for Nonlinear Optimization," Sandia Report SAND94-8225, Sandia National Laboratories, Livermore, CA, March 1994.

[5]Hart, W.E., "SGOPT, A C++ Library of (Stochastic) Global Optimization Algorithms," Sandia Technical Report SAND98-XXXX, In preparation.

[6]Byrd, R.H., Schnabel, R.B., and Schultz, G.A., "Parallel quasi-Newton methods for unconstrained optimization," *Math. Prog.*, 42(1988), pp. 273-306.

[7]Byrd, R.H., Schnabel, R.B., and Schultz, G.A., "Using parallel function evaluations to improve Hessian approximation for unconstrained optimization," *Ann. Oper. Res.*, 14(1988), pp. 167-193.

[8]Dennis, J.E., and Torczon, V.J., "Derivative-Free Pattern Search Methods for Multidisciplinary Design Problems," paper AIAA-94-4349 in *Proceedings of the 5th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Panama City, FL, Sept. 7-9, 1994, pp. 922-932.

[9]Huang, M.-W., Arora, J.S., "Optimal Design with Discrete Variables: Some Numerical Experiments," *International Journal for Numerical Methods in Engineering*, 40(1997), pp.165-188.

[10]http://cfd.sandia.gov/Pronto3D.html

[11]http://cfd.sandia.gov/docs/coyote/coyote-welcome.html

[12]http://www.cs.sandia.gov/CRF/MPSalsa/

[13]http://www.sandia.gov/1431/ALEGRAw.html

[14]http://www.sandia.gov/1431/PCTHw.html

[15]http://cfd.sandia.gov/sierra.html

[16]Gropp, W., Lusk, E., and Skjellum, A., *Using MPI, Portable Parallel Programing with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

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

[18]http://www.acl.lanl.gov/~pfay/teraflop/

[19]Shadid, J.N., and Tuminaro, R.S., "Sparse iterative algorithm software for large-scale MIMD machines: an initial discussion and implementation," *Concurrency: Practice and Experience*, Vol. 4(6), Sept. 1992, pp. 481-497.

[20]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 Multidisciplinary Analysis and Optimization*, Bellevue, WA, Sept. 4-6, 1996, pp. 1568-1582.

[21]Shadid, J.N., Moffat, H.K., Hutchinson, S.A., Hennigan, G.L., Devine, K.D., and Salinger, A.G., *MPSalsa - A finite element computer program for reacting flow problems. Part 1 - Theoretical development.* Sandia Technical Report SAND95-2752, Albuquerque, NM, 1996.

[22]Salinger, A.G., Devine, K.D., Hennigan, G.L., Moffat, H.K., Hutchinson, S.A., and Shadid, J.N., *MPSalsa - A finite element computer program for reacting flow problems. Part 2 - User's Guide*, Sandia Technical Report SAND96-2331, 1996.

[23]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.

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

[25]http://mephisto.ca.sandia.gov/TFLOP/sc96/index.html

[26]"MPI-2: Extensions to the Message-Passing Interface," Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee, July 18, 1997.