High-performance numerical optimization on multicore clusters

Panagiotis E. Hadjidoukas[1], Constantinos Voglis[1], Vassilios V. Dimakopoulos[1], Isaac E. Lagaris[1], and Dimitris G. Papageorgiou[2]

[1] Department of Computer Science
{phadjido,voglis,dimako,lagaris}@cs.uoi.gr
[2] Department of Materials Science and Engineering
dpapageo@cc.uoi.gr
University of Ioannina, Ioannina, Greece, GR-45110

Abstract. This paper presents a software infrastructure for high performance numerical optimization on clusters of multicore systems. At the core, a runtime system implements a programming and execution environment for irregular and adaptive task-based parallelism. Building on this, we extract and exploit the parallelism of a global optimization application at multiple levels, which include Hessian calculations and Newton-based local optimizations. We discuss parallel implementations details and task distribution schemes for managing nested parallelism. Finally, we report experimental performance results for all the components of our software system on a multicore cluster.

Keywords: task parallelism, message passing, numerical differentiation, global optimization

1 Introduction

Numerical optimization is a useful tool that has been widely used on many scientific problems such as space trajectory calculation and computation of optimal shapes for automobile or aircraft components. Optimization problems, especially global ones, have high computational demands because of the substantial execution time and the possibly multiple local mimima of the objective function to minimize. Exploitation of parallelism at several levels such as function evaluations, numerical computations and the optimization algorithms themselves can drastically reduce the time required to find a solution.

The *Multistart* method is a standard and widely used scheme for dealing with global optimization problems. According to this method, a local optimization procedure is applied to a number of randomly selected points. Local optimization can be based on the Newton method with Hessian modification, a general and powerful method for multidimensional non-linear optimization that makes use of first and second derivatives of the objective function. This, in turn, introduces further computational complexity as derivative estimation via finite differentiation requires a number of function evaluations.

Task-based parallelism, as expressed by the master-worker programming paradigm, can be an effective approach for a cluster-aware implementation of global optimization methods such as Multistart. Function evaluations are mapped to tasks and assigned to the workers. The dynamic load balancing of the model further enhances its suitability. A naive implementation of the model, however, cannot meet all the requirements that Multistart imposes. First, the large expected number of spawned tasks (typically on the order of 10^6) affect the scalability as the single master becomes a bottleneck. Secondly, the exploitation of nested parallelism requires advanced runtime techniques, able to provide efficient management of processing elements. Additionally, it is important to have a hardware-independent solution that transparently uses multi-threading to fully exploit the physically shared memory of SMP/multi-core systems.

In this paper, we present a software infrastructure that deals with all the above limitation issues that concern the parallelization of the Multistart method. At the core of the system there is TORC, a novel runtime environment for programming and executing irregular and adaptive master-worker applications on multi-core SMPs and clusters of such machines. As such, TORC targets both message passing and shared memory programs by exporting an API that provides ease of programming and transparent load balancing without requiring any interaction with the low-level message passing primitives. Building on TORC, we design a standalone numerical differentiation software package (PNDL) that provides routines for gradient and Hessian computations. We manage to extract parallelism at all possible levels in a straightforward and seamless manner, while we present several task distribution schemes, which are combined with the work stealing mechanisms of TORC. Finally, we present the parallelization of a Newton-based Multistart method using both TORC and PNDL to execute multiple local optimizations and gradient/Hessian calculations. The experimental evaluation on a dedicated multicore cluster demonstrates the efficiency of our system.

The rest of this paper is organized as follows: Section 2 gives a brief introduction to the non-linear global optimization problem. Section 3 discusses the parallelization issues of Multistart. Sections 4 and 5 present the TORC tasking library and PNDL. Experimental evaluation and related work are reported in Sections 6 and 7 respectively. We conclude with a discussion in Section 8.

2 Numerical optimization

The task of numerical optimization is to locate (approximate) a minimizer of a generally multidimensional objective function. The mathematical formulation is

$$\min_{x \in \mathbb{R}^n} f(x) \tag{1}$$

where $x \in \mathbb{R}^n$ is a real vector and $f : \mathbb{R}^n \to \mathbb{R}$ the objective function. There exist a plethora of applications in physics, chemistry, engineering, and economics that can be formulated as optimization problems. Predicting the tertiary protein structures, defining optimal sea routes, calculating bound states for few body systems, tuning all kinds of machine learning models and identifying the seismic properties of a piece of the earths crust, are all examples of real world applications that can be tackled as optimization problems.

An optimization algorithm is a sequential procedure that, beginning from a starting point $x_0 \in S$, generates a sequence of iterates $\{x_k\}_{k=0}^{\infty}$ that terminates when the solution point is approximated with a prescribed accuracy. In deciding how to move from one iterate x_k to the next the algorithm uses information about the function at x_k (function value, first or second order derivatives). A general class of optimization algorithms use second order derivative information of the objective function and use it to build and minimize a quadratic model around the current iteration. The main representative of this class is the *Newton* method. At each iterate, the Newton method makes use of first and second order derivative information to proceed to the next point. This can be achieved using a *line search* algorithm which searches along a descent direction $p_k \in \mathbb{R}^n$ for an iterate with lower function value. The distance to move along p_k can be found by solving the following one-dimensional minimization problem that is to find a step length α that minimizes $f(x_k + \alpha p_k)$. The main computational cost of a single Newton iteration is determined by the objective function and the derivatives calculation that are used to compute the search direction.

In many cases derivatives cannot be expressed analytically because the underlying functions are represented by large and complicated computer codes. In these cases finite differencing is an approach for calculating the first and second order derivatives of an $n-$dimensional objective function at a point x by examining the objective function behavior on small finite perturbations around x. The number of function evaluations depends on the order of the derivative (first or second) and on the requested accuracy (the larger accuracy the more function evaluations). For the gradient vector at least $n+1$ function evaluations are required and for the Hessian at least $n(n+1)/2$. Two of the most popular formulas for approximating gradient and Hessian, using central differences are summarized below:

$$\begin{aligned} \frac{\partial f(x)}{\partial x_i} &\approx \frac{f(x+\epsilon e_i) - f(x-\epsilon e_i)}{2\epsilon} \\ \frac{\partial^2 f(x)}{\partial x_i \partial x_j} &\approx \frac{f(x+\epsilon e_i+\epsilon e_j) - f(x-\epsilon e_i+\epsilon e_j)}{4\epsilon^2} - \frac{f(x+\epsilon e_i-\epsilon e_j) + f(x-\epsilon e_i-\epsilon e_j)}{4\epsilon^2} \end{aligned} \tag{2}$$

where e_i is the $i-$th unit vector and ϵ a small positive scalar. Finite differencing is a perfect candidate for parallel execution. All function evaluations in Eq.(2), $f(x+\epsilon e_i)$ and $f(x+\epsilon e_i+\epsilon e_j)$, can be performed independently and in parallel.

The Newton procedure locates a minimizer efficiently with quadratic convergence speed. However, there is no guarantee that this minimizer will be the one with the lowest function value in all S, as the minimizer may stick at a local minimum. This requirement introduces the problem of *global optimization*, one of the most difficult problems in applied mathematics. Searching for the global minimum is a quite challenging, yet extremely useful task for all applications mentioned in the beginning of this Section. It is proven, in the multidimensional case, that it is impossible to guarantee the globally optimal value will be found in finite time. All that can be assured is that the probability of locating the global minimizer approximates 1. One of the oldest and most popular schemes for deal-

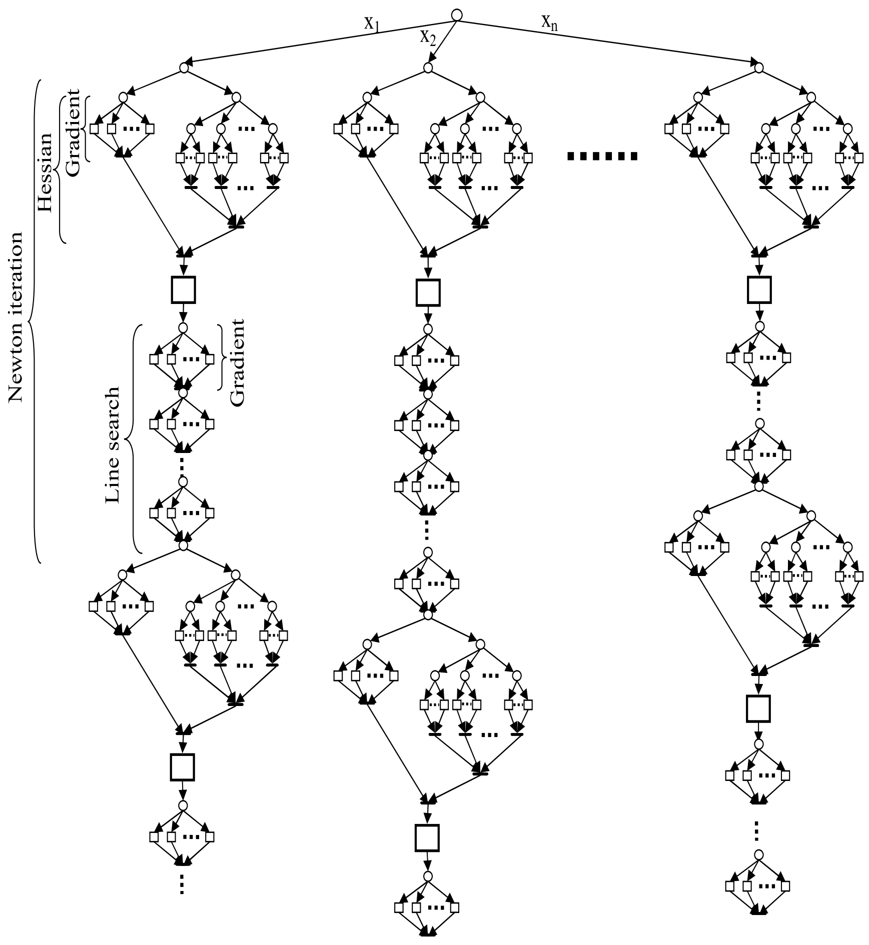

Fig. 1. Execution task graph for Multistart using finite difference derivatives

ing with global optimization problems is the *Multistart* method. According to this method, a local search procedure $\mathcal{L}$ is executed for each point in a sample generated from a uniform distribution over the search space S. Albeit simple in principle, *Multistart* is the heart of more sophisticated global optimization algorithms such as *clustering methods*[1, 2].

3 Multistart parallelism issues

In the framework of global optimization based on numerical differentiation, there exist several levels of parallelism that can be exploited in order to accelerate the method. Fig. 1 illustrates the execution task graph of the Multistart method. Each circle corresponds to code that spawns parallelism, which can be expressed and instantiated with lower-level tasks. Tasks at the innermost level are represented with squares and correspond to serial code and specifically to either single function evaluations or sequential direct linear algebra operations. Therefore, the paths of the graph represent operations that can be performed in parallel while their meeting points represent the completion of all tasks in a team with the satisfaction of all data and control dependencies.

Initially, the application runs the Multistart method and spawns first-level ($L1$) tasks. These perform the Newton *local search* method to multiple independent initial points (x_i) and execute iterations until the convergence criterion is met. In each iteration, the tasks first proceed with the derivative calculation, spawning two second-level ($L2$) tasks that compute the gradient and Hessian respectively. The gradient computation includes a number of function evaluation ($L3$) tasks. The Hessian computation, however, exploits an additional level of parallelism by assigning the numerical calculation of each partial derivative to a ($L3$) task that can spawn two to nine function evaluation ($L4$) tasks, depending on the desired accuracy and the bounds. Local search continues with a sequential task that performs the required matrix modification and the solution of the linear system. The iterative line search method follows, exploiting each time a single level of parallelism for the gradient computation. For a large number of initial points, a gradual execution of Multistart can be performed by applying the Newton method to bunches of points. In such case, the execution task graph is repeated until the desired number of points has been processed.

Multistart is a highly irregular parallel application: first, the local search method is applied concurrently to multiple points, the number of which may not be exactly divided by the number of available processors. Secondly, the execution time of local search exhibits significant variation as the number of iterations required for convergence depends on the randomly selected initial point. Similarly, the line search method is performed for an unknown before number of iterations. Irregularity is found even at the innermost level of parallelism (Hessian calculation), as the number of function evaluations for the derivative computation at a specific point also depends on the imposed bounds on the variables. According to the above, the execution times for finding a minimum for each initial point are neither balanced nor known beforehand. Derivative estimation via finite differencing is computationally expensive for several applications where the time for a single function call is substantial. Therefore, the highly irregular nested parallelism of Multistart must be exploited at all possible levels, without making any assumption about the number of available processors.

4 TORC runtime library

TORC [3] implements a task-based programming and runtime environment that makes the development of master-worker applications almost trivial. Although TORC supports several features, due to lack of space we briefly present only those related to the parallelization of the Multistart method.

TORC assumes that a single application consists of multiple MPI processes with private memory when running on the cluster. Furthermore, it uses multi-threading to exploit the multiprocessor/multicore cluster nodes. A *task* represents a work unit that is independent of its execution vehicle, i.e. the MPI process or thread. A spawned task can be submitted for execution to *any* MPI process; the programmer may specify the target process in the task creation routine. When a parent task blocks, its underlying vehicle can proceed to the execution of other ready-to-run tasks. This means that a TORC application can run successfully even if only a single-threaded process is used. Furthermore, each process can have multiple worker threads. Therefore, the same application code can run on any combination of MPI processes and threads, exploiting at runtime the presence of physically shared memory, if available.

Due to the decoupling of tasks and execution vehicles, multiple levels of task parallelism are inherently supported and any child task can become a master and spawn new tasks. Therefore, TORC enables the programmer to express hierarchical and recursive task parallelism naturally, which would be otherwise quite difficult to implement. In the task creation routine (`torc_task()`), the user specifies the task function, the number of arguments this function receives and an argument list. For each argument, its size and data type is required. In addition, an intent attribute must be also supplied, similarly to the IN, OUT and INOUT intent attributes of Fortran 90. Any data movement is performed transparently to the user. After task creation, a master task calls the `torc_waitall()` routine to suspend itself until all child tasks have finished and their results have arrived. Several master-worker applications may have global data that is initialized by the master and then broadcast to the workers. The `torc_bcast()` routine allows any task to broadcast global data to all MPI processes, thus avoiding unnecessary data transfers.

As task stealing is inherently supported by TORC, the programmer has only to decide about the task distribution scheme, by querying the execution environment and then specifying the node or worker where each task will be initially submitted for execution. The scheduling loop of a worker thread is activated when its current task finishes or blocks. A worker extracts and executes the task that is at the front of its local ready queue. If this is empty, the worker tries to steal a task from the rest of the intra-node ready queues. If inter-node task stealing is enabled, it issues requests for work to remote nodes in sequential order. Task stealing is always performed from the back of the ready queues. The stealing of a task from a remote queue includes the corresponding data movement, unless the task returns to its parent node. Inter-node task stealing is optional and must be explicitly enabled based on the load imbalance of the parallel application. On the other hand, intra-node task stealing is always active. A more detailed description of the TORC library can be found in [3].

5 PNDL and parallel Multistart implementation

The parallel implementation of the numerical differentiation library for multicore clusters has been based on the tasking model that TORC provides. For each function evaluation, a task is created, with main input argument a vector x and

```
! first level
subroutine pndlhf(f, x, n, iord, hes)
external f, driver
integer n, iord
double precision x(n), hes(n,n), xx(n)
common /data/ xx
...
<set xx(I) = x(I)> ! create copy of x
call torc_bcast(xx, n, MPI_DOUBLE_PRECISION)
iworker = torc_worker_id()
nworkers = torc_num_workers()
<for each derivative>
  call torc_task(iworker, driver, ..)
  istride1 = <# function values required>
  iworker = mod(iworker+istride1,nworkers)
call torc_waitall()
end
```

```
! second level
subroutine driver(f, n, ...)
double precision xx(n)
common /data/ xx(n)
...
iworker = torc_worker_id()
nworkers = torc_num_workers()
istride2 = 1
<for each required function value>
  call torc_task(iworker, f, ..)
  iworker = mod(iworker+istride2,nworkers)
call torc_waitall()
<compute partial derivative h(i,j)>
end
```

Fig. 2. Outline of a PNDL Hessian calculation with exploitation of two levels of parallelism using the `STRIDE` distribution scheme

result the computed function value $f(x)$. The core routine that PNDL implements for Hessian computations is: `pndlhf(f,x,n,iord,hes)`, where f is the function to be differentiated, x the vector containing the point of calculation, n the dimensionality of the function, *iord* the requested order of accuracy, and *hes* the resulting Hessian matrix.

The parallel routines that PNDL exports to MPI programs have been redesigned for a master-worker execution mode. The calling process initializes the input parameters and receives the computed derivatives. When a PNDL routine is invoked, the primary task initially broadcasts the input vector x, through the use of a common block. If the routine has a single level of parallelism, function evaluation tasks are spawned and distributed cyclically to the workers. After task completion, the primary task uses the gathered function values to compute the derivatives. Although a reduction operation can be used, the adopted scheme preserves the sequential order of calculations and, thus, avoids rounding errors.

The above scheme, however, may increase significantly the memory requirements of PNDL for the estimation of second order derivatives of functions with a large number of variables, which can be of the order of thousands for specific problems. To handle this issue, we exploit nested parallelism; each element of the Hessian is calculated by a first-level task, which issues function calls through second-level tasks. The number of first-level tasks is equal to $(n(n+1)/2)$ and each of them spawns 2 to 9 second-level tasks, according to user parameters. Memory usage is drastically reduced because the number of *active* first-level tasks, which reserve stack space for the results, never exceeds the number of available workers. This is achieved because second-level tasks are inserted in the front of the ready queues and thus have higher execution priority than first-level tasks, which are inserted at the end.

The runtime architecture of TORC allows for several task distribution schemes: Fig. 2 outlines the hierarchical parallel implementation of the `pndlhf` routine using the `STRIDE` scheme, which divides equally the number of function evaluations among the available workers. The first argument of the task creation routine denotes the identifier of the worker thread where the task will be submitted to. The parent task distributes the first-level tasks using a variable stride (`istride1`) that is determined by the (known beforehand) number of second-level tasks that correspond to each task. Next, each first-level task distributes the inner tasks to consecutive workers (`istride2=1`), starting from the worker where that task runs on. The `STRIDE` scheme is, however, suitable only for dedicated homogeneous clusters and may result in an excessive number of messages for high-dimensional functions. To overcome these issues we have introduced a dynamic task distribution scheme, called GLTS, which distributes the first-level tasks cyclically across the processors (`istride1=1`) and submits the second-level tasks locally (`istride2=0`) with task stealing enabled. GL is another task distribution scheme that differs from GLTS in that task stealing is used only at the intra-node level, i.e. between workers that belong to the same process. Finally, LLTS is a variant of the GLTS scheme that submits even the first-level tasks locally and specifically in the ready queue of the worker that issued the PNDL routine (both strides are equal to zero).

The parallelization of the Newton method relies on two PNDL routines that compute the required gradient and Hessian matrices. These routines can be executed concurrently, as an additional level of parallelism, through the spawning of two TORC tasks. This, however, requires appropriate modifications in PNDL, due to the usage of the common block for broadcasting the input vector. Therefore, we use an array of input vectors in the common block and each PNDL function call is dynamically assigned a unique identifier that specifies an available entry of this array. Parallel Multistart takes advantage of the reentrancy of PNDL functions to issue multiple local searches concurrently starting from randomly chosen initial points. As the number of points increases, the small serial fraction of the Newton method becomes negligible and the effective utilization of parallel hardware is further improved. For Multistart, we have followed MLTS, a modified LLTS distribution scheme: tasks at the first-level of parallelism, i.e. Newton optimizations, are distributed cyclically across the workers. Parallelism at all inner levels is submitted locally, with inter-node task stealing enabled. Ideally, each local search will be performed exclusively by a single worker. Idle workers will try to steal and execute tasks that belong to the first-level of parallelism and will participate in the execution of remotely issued PNDL routines only when the number of remaining optimizations is less than the number of workers.

6 Performance experiments

In this section we present experimental results from application executions on a dedicated 16-node Sun Fire x4100 cluster with Gigabit Ethernet, each node with 2 dual-core AMD Opteron 275 CPUs. The software setup includes Linux 2.6, GCC 4.3 and MPICH2. In all experiments we use the multithreaded configuration, running a single process with multiple workers on each cluster node.

Our system targets mostly medium to coarse-grained tasks for remote execution. As an indication, for the specific platform used for our experimental evaluation, the task execution overhead is measured approximately 0.1ms for a zero-argument task; this overhead however decreases with the number of tasks due to the overlap of task creation, data movement and task execution. Thus, the overhead for the single task case depends on the latency of the interconnection network, while the overall minimum overhead depends on the maximum bandwidth. In addition, the minimum overhead for a given number of tasks is a linear function of the argument size. In contrast, within a multi-core node we support very fine-grained tasks efficiently.

Parallel Hessian. We present two sets of synthetic experiments that calculate the Hessian with $O(h^4)$ precision without imposing bounds on the variables. The first set of experiments (E1, E2) uses a test function with 20 variables and leads to a total of 820 objective function calls. We have arranged for function evaluation time to be 100ms and 1000ms via appropriate artificial delays. The second set (E3, E4) uses a 100-dimensional test function with artificial delays of 10ms and 100ms. The number of function evaluations for this set is 20200. Both experiments are designed to cover a wide range of practical situations and correspond to medium and large problem sizes. They are representative of applications with many dimensions and/or substantial function execution time.

Figs. 3-4 and 5-6 present the results from the two sets of experiments with the 4 task distribution schemes (STRIDE, GLTS, GL, and LLTS). For the first set, we observe that the speedup increases with the computational cost of the test function, due to the higher computation-to-communication ratio. The slight decrease in performance is attributed to several factors: a small serial fraction of code in the PNDL function, the overhead for broadcasting the point and the load imbalance when the number of function evaluations is not exactly divided by the number of workers. Although all distribution schemes exhibit comparable performance up to 32 processors, GLTS achieves the highest speedup on 64 processors, with LLTS and GL to follow. The lowest speedup corresponds to the STRIDE scheme because of its large number of explicit messages. For the 100-dimensional function (Figs. 5 and 6), the obtained speedup of GLTS almost coincides with the ideal for both cases. In this set of experiments, the lowest speedup values are observed for LLTS, due to the bottleneck at the single queue where the 5050 first-level tasks are submitted for execution.

Parallel Multistart. In order to evaluate parallel Multistart we use a test function with 10 variables, artificial delays that range from 1ms to 1s, and the modified LLTS task distribution scheme. Fig. 7 depicts the speedup for a single starting point, which represents a worst-case but unlikely to occur scenario in global optimization problems. We observe that the Newton method fails to scale as the number of workers increases, regardless of the function evaluation time. This is mostly attributed to the small sequential task ($\simeq$2%) of the Newton method. The speedup can be further affected by the communication overheads, especially when the computational cost of the objective function is low. For function evaluation time equal to 1s, however, the measured speedup is very close to the maximum theoretical speedup as defined by Amdahl's law. Figs. 8 to 10 show the speedup of Multistart for 16, 64 and 1024 optimizations. The attained

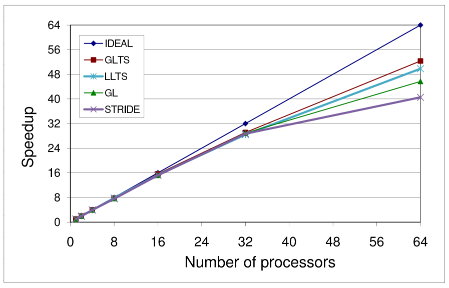

Fig. 3. Speedup for experiment E1 (variables=20, delay=100ms)

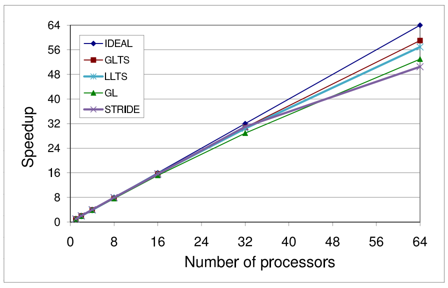

Fig. 4. Speedup for experiment E2 (variables=20, delay=1000ms)

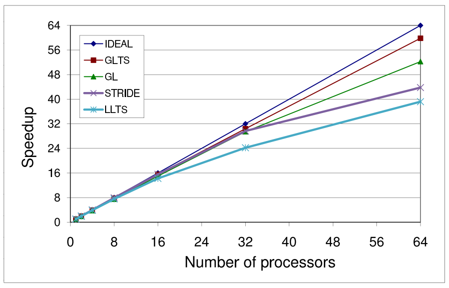

Fig. 5. Speedup for experiment E3 (variables=100, delay=10ms)

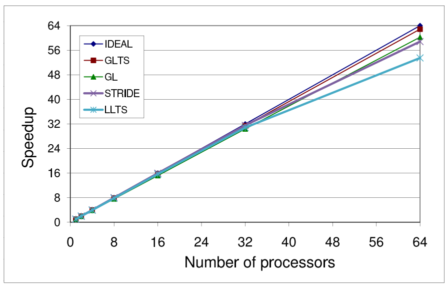

Fig. 6. Speedup for experiment E4 (variables=100, delay=100ms)

speedup increases with the number of optimizations, especially if this exceeds the number of available processing cores. For 1024 optimizations, the speedup almost coincides with the ideal for both 10ms and 100ms evaluation time. The performance results are in accordance with those obtained for a real application case that deals with the protein folding problem [4].

7 Related work

Although many parallel local and global optimization algorithms were proposed in the last decades (e.g. [5, 6], only a handful of actual systems exist. One of the most widely used scientific software programs, MATLAB, presented its first parallel optimization solution in 2009 [7]. In the pioneer work of [8] an interval global optimization method is implemented using dynamic load balancing. PGO [9] is a general parallel computing based on the Genetic Algorithm. In PGO, the parallel (and heterogeneous) computing framework is organized as a global master-slave system using a central database management system for storing all the data during optimization progress. Oriented in interoperability, the MHGrid platform [10] exploits meta-heuristics based search methods and Grid computing to enable the transparent sharing of heterogeneous and dynamic resources offering a versatile Global optimization framework. MANGO [11] is a middleware that involves the

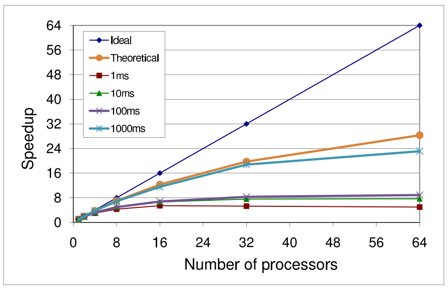

Fig. 7. Speedup for 1 local search

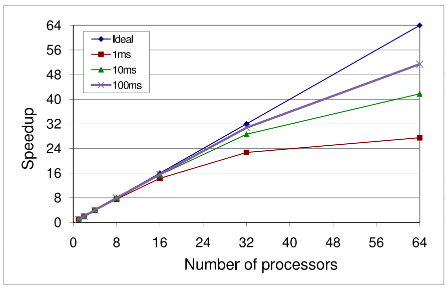

Fig. 8. Speedup for 16 local searches

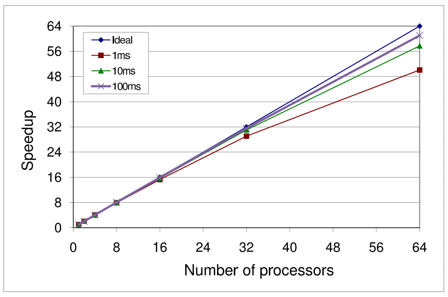

Fig. 9. Speedup for 64 local searches

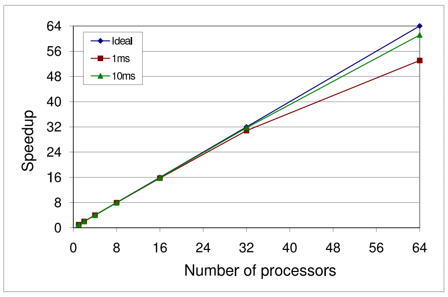

Fig. 10. Speedup for 1024 local searches

development of an extensible and flexible multiagent platform, in which autonomous agents can solve global optimization problems in cooperation. Finally, PaGMO[12] is a open source multi-threaded software that offers a plethora of local and global optimization codes exploiting modern multi-core architectures. In contrast to our infrastructure, none of the above supports hierarchical and multi-level task parallelism. In addition, our system is platform-agnostic supporting transparently both shared and distributed memory architectures.

Despite the availability of several software packages for estimating derivatives numerically (e.g. [13, 14]) their implementation is sequential. The only parallel numerical Hessian implementations we are aware of are [15] and [16], mainly used for computational chemistry.

8 Conclusions

We presented a system for efficient exploitation of nested and irregular parallelism in non-linear optimization problems. At the core of our system is TORC, a runtime library that supports adaptive task-based parallelism on clusters of multicores/SMPs. Using TORC, we manage to extract and execute the multiple levels of parallelism inherent in the Multistart optimization method, performing thus Newton-based local searches, gradient and Hessian calculations and function evaluations in parallel.

Our ongoing work includes the integration of additional numerical optimization techniques into our infrastructure. We also work on the efficient parallelization of a real application case, concerning the protein folding problem. Finally, we plan to extend the applicability of our system to computational grids and GPGPU environments.

References

1. Rinnooy Kan, A.H.G., Timmer, G.T.: Stochastic global optimization methods part I: Clustering methods. Math. Progr. 39, 27–56 (1987)
2. Voglis, C.,Lagaris, I.E.: Towards ideal multistart, A stochastic approach for locating the minima of a continuous function inside a bounded domain. Applied Math. and Comput. 213, 216–229 (2009)
3. Hadjidoukas, P.E., Dimakopoulos, V.V.: TORC: a tasking library for multicore clusters. Technical Report TR-2011-6, Dept. of Computer Science, University of Ioannina, Greece (2011)
4. Voglis, C., Hadjidoukas, P.E., Dimakopoulos, V.V., Lagaris, I.E., Papageorgiou, D.G.: Task-parallel global optimization with application to protein folding. In: 9th Int'l Conf. on High Perf. Comput. and Simul., Istanbul, Turkey (2011).
5. Schutte, J.F., Reinbolt, J.A., Fregly, B.J., Haftka, R.T., George, A.D.: Parallel global optimization with the particle swarm algorithm. Int'l J. Numerical Methods in Engin. 61(13), 2296–2315 (2004)
6. Byrd, R.H., Eskow, E., van der Hoek, A., Schnabel, R.B., Oldenkamp, K.P.B.:A Parallel global optimization method for solving molecular cluster and polymer conformation problems. In: 7th Siam Conf. on Parallel Processing for Scientific Comput., pp. 72–77, SIAM Philadelphia (1995)
7. Kozola, S.: Improving optimization performance with parallel computing. MATLAB Digest (2009)
8. Eriksson, J., Lindstrom, P.: A parallel interval method implementation for global optimization using dynamic load balancing. Rel. Comput. 1/2, 77–91 (1995)
9. He, K., Zheng, L., Dong, S., Tang, L., Wu, J., Zheng, C.: PGO: a parallel computing platform for global optimization based on genetic algorithm. Computers & Geosciences 33(3), 357–366 (2006)
10. Wahib, M., Munetomo, M., Munawar, A., Akama, K.: Mhgrid: Towards an ideal optimization environment for global optimization problems using grid computing. In: 8th Int'l Conf. on Parallel and Distr. Comput., Applic. and Technologies, pp. 167-168, IEEE Computer Society, Washington, DC (2007)
11. Günay, A., Öztoprak, F., Birbil, Ş, Yolum, P.: Solving global optimization problems using MANGO. In: 3rd KES Int'l Symp. on Agent and Multi-Agent Systems: Technologies and Applic., pp. 783-792, Upsalla, Sweden (2009)
12. Biscani, F., Izzo D., Yam, C.: A global optimisation toolbox for massively parallel engineering optimisation. In: 4th Int'l Conf. on Astrodynamics Tools and Techniques, Madrid, Spain (2010)
13. GSL, GNU Scientific Library. `http://www.gnu.org/software/gsl/` (2010)
14. NAG Fortran Library, D04 Numerical Differentiation, subroutine D04AAF
15. Krishnan, M., Alexeev, Y., Windus, T.L., Nieplocha, J.: Multilevel parallelism in computational chemistry using Common Component Architecture and Global Arrays. In: ACM/IEEE Supercomp. Conf., p.23, Seattle, WA (2005)
16. Staveley, M.S., Poirier, R.A., Bungay, S.D.: An evaluation of parallel numerical Hessian calculations. In: High Perf. Comput. Symp., pp. 196-214, Kingston, ON, Canada (2009)