Applied Mathematics and Computation 231 (2014) 544–559

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

Supporting adaptive and irregular parallelism for non-linear numerical optimization

P.E. Hadjidoukas [a,*], C. Voglis [b], V.V. Dimakopoulos [b], I.E. Lagaris [b], D.G. Papageorgiou [c]

[a] Chair of Computational Science, ETH Zurich, Zurich CH-8092, Switzerland
[b] Department of Computer Science, University of Ioannina, GR-45110 Ioannina, Greece
[c] Department of Materials Science and Engineering, University of Ioannina, GR-45110 Ioannina, Greece

ARTICLE INFO

Keywords:
Irregular and multilevel parallelism
Adaptive task parallelism
Multicore clusters
Message passing
Numerical differentiation
Numerical optimization
Protein conformation

ABSTRACT

This paper presents an infrastructure for high performance numerical optimization on clusters of multicore systems. Building on a runtime system which implements a programming and execution environment for irregular and adaptive task-based parallelism, we extract and exploit the parallelism of a Multistart optimization strategy at multiple levels, which include second order derivative calculations for Newton-based local optimization. The runtime system can support a dynamically changing hierarchical execution graph, without any assumptions on the levels of parallelization. This enables the optimization practitioners to implement, transparently, even more complicated schemes. We discuss parallelization details and task distribution schemes for managing nested and dynamic parallelism. In addition, we apply our framework to a real-world application case that concerns the protein conformation problem. Finally, we report performance results for all the components of our system on a multicore cluster.

1. Introduction

Numerical optimization is an indispensable tool that has been widely applied on many scientific problems. An important feature of optimization problems is their long execution time which is attributed to the high computational demands and the possibly multiple local minima of the objective function to minimize. There are several applications where the time for a single function call is substantial. Parallelization can drastically reduce the required processing time to find a solution. The inherent parallelism of these problems can be found at various levels, including function or gradient evaluations, linear algebra calculations and the optimization algorithms themselves. Global optimization algorithms that can take advantage of parallel and distributed architectures are particularly suitable for solving problems with high computational requirements. The emerging multi-core architectures provide a cost-effective solution for high-performance optimization.

Although many parallel local and global optimization algorithms were proposed in the last decades [1–6], 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. In [9] the authors present ParaGlobSol, a parallel global optimization package written in Fortran 90/95 with MPI to perform inter process communication and a dynamic load balancing scheme. PGO [10] 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

* Corresponding author.
E-mail addresses: phadjido@mavt.ethz.gr (P.E. Hadjidoukas), voglis@cs.uoi.gr (C. Voglis), dimako@cs.uoi.gr (V.V. Dimakopoulos), lagaris@cs.uoi.gr (I.E. Lagaris), dpapageo@cc.uoi.gr (D.G. Papageorgiou).

0096-3003/$ - see front matter © 2014 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.amc.2013.12.092

progress. Oriented in interoperability, the MHGrid platform [11] 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 [12] is a middleware that involves the development of an extensible and flexible multiagent platform, in which autonomous agents can solve global optimization problems in cooperation. PaGMO [13] is a recently released open source multi-threaded software that offers a plethora of local and global optimization codes exploiting modern multicore architectures. Finally in [2,14] the authors present VTDIRECT95 a parallel implementation of the DIRECT algorithm, using a three-level hierarchical parallel scheme.

In this work we study the parallelization of Multistart [15] method which 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. The Multistart method forms the basis for many successful global optimization methods and any parallelization study on it can be easily extended to more elaborate optimization schemes. For local optimization we have chosen the Newton method which is a general and powerful method for multidimensional non-linear optimization that makes use of first and second derivatives of the objective function. This choice, in turn, introduces further computational complexity as derivative estimation via finite differentiation requires a number of function evaluations. In many practical situations analytical calculation of second order derivatives may be prohibitive.

Task-based parallelism, as expressed by the master-worker programming model, 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 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 a parallel global optimization method (Multistart) imposes. First, the large expected number of spawned tasks affect scalability as the single master becomes a bottleneck. Secondly, the exploitation of nested parallelism requires advanced runtime techniques, able to facilitate programming and provide efficient management of processing elements. Ideally, a parallel implementation must target both shared and distributed memory machines, without the burden of explicit message passing for the programmer. Additionally, it is important, from a performance point of view, to have a hardware-independent solution that transparently uses multi-threading to fully exploit the physically shared memory of SMP/multi-core systems.

We first present our Tasking library for Clusters (TORC), a novel general-purpose runtime environment for programming and executing irregular and adaptive task-parallel applications on multi-core SMPs and clusters of such machines. Building on TORC, we design a standalone Parallel Numerical Differentiation Library (PNDL) that provides routines for first and second order derivative approximation. For the latter we extract two levels of parallelism and study several task distribution schemes.

We also present the parallelization of a Newton-based Multistart method using both TORC and PNDL to execute concurrently multiple local optimizations and gradient/Hessian calculations. In contrast to previous approaches, we manage to express and exploit parallelism at all possible levels in a straightforward and seamless manner using a single runtime framework. In addition, the task distribution and stealing mechanisms of TORC provide efficient load balancing without the need for explicit partitioning of processors.

In contrast to other infrastructure, and with the exception of VTDIRECT95, our proposal is the only one that supports hierarchical and multi-level task parallelism. What differentiates our approach from VTDIRECT95 implementation, is that our hierarchical execution graph is changing dynamically unlike the static three level scheme applied in [14]. In addition, our system is platform-agnostic supporting transparently both shared and distributed memory architectures.

By integrating a molecular modeling software package [16] with our system, we are able to apply our framework to a real application case that deals with the protein conformation problem, that is the problem of determining the three dimensional structure of a protein. It is a fundamental problem in biophysical sciences with applications in drug design and in genetic information decoding.

We present an experimental evaluation on a dedicated homogeneous multicore cluster, providing results at both intranode and cluster-wide levels using a single application executable. The performance results with synthetic benchmarks and applications demonstrate the efficiency of our system.

Summarizing, the contributions of this paper are:

- a runtime library for task-based computations of multicore clusters, which allows for extraction and exploitation of dynamic, hierarchical and multilevel parallelism in global optimization algorithms,
- a novel high-performance implementation of the Multistart method using the above infrastructure, over different architectures, which allows multiple numerical derivative computations to be deployed concurrently,
- a highly efficient application of the optimization algorithm involved in the protein folding problem.

The rest of this paper is organized as follows: Section 2 gives an overview of the non-linear global optimization problem. Section 3 discusses the inherent parallelism structure of Multistart and introduces the organization of our software infrastructure. Section 4 outlines the programming interface and some implementation details of TORC. Section 5 focuses on PNDL and Multistart, while Section 6 presents the protein conformation problem. Experimental evaluation is reported in Section 7. We conclude with a discussion in Section 8.

2. Numerical optimization

2.1. Optimization of a non-linear objective function

The case of non-linear optimization, deals with non-linear objective functions that depend on real variables, with bound restrictions on the values of these variables. The mathematical formulation is

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

where $x \in S \subseteq \mathbb{R}^n$ is a real vector and $f : \mathbb{R}^n \to \mathbb{R}$ a smooth function.

An optimization algorithm is a sequential procedure that, beginning from an 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 optimization algorithm uses information about the function at x_k (function value, first or second order derivatives). The goal is to find a next iterate x_{k+1} with a lower function value than x_k.

One of the most successful iterative algorithms for the problem in Eq. 1 is the *Newton method*. This method uses first and second order derivative around a current iterate x_k to define a positive definite quadratic model and attempts a step towards the minimum of this model. In regions near a minimum the quadratic model is expected to match the objective function. The quadratic model at k-th iteration is defined as

$$q_k(p) \approx f(x_k) + p^T g_k + \frac{1}{2} p^T B_k p,$$

where $g_k = \nabla f(x_k)$ the vector of first order derivatives and $B_k = \nabla^2 f(x_k)$ the *Hessian* matrix of second order derivatives. Assuming that B_k is positive definite, the minimum of the model is defined as:

$$p_k = -B_k^{-1} g_k. \tag{2}$$

Procedure. `Newton` (f, x_0, x^*)

Input: Objective function, $f : S \subset \mathbb{R}^n \to \mathbb{R}$; starting point: x_0
Output: Approximation of the local minimizer: x^*

1 **for** $k \leftarrow 1, 2, \ldots$ **do**
2 Calculate $g_k = \nabla f(x_k)$ and $B_k = \nabla^2 f(x_k)$
3 Factorize the matrix G_k, where $G_k = B_k$ if B_k is positive definite; otherwise, $G_k = B_k + E_k$
4 Solve $G_k p_k = -g_k$ to obtain p_k
5 Set $x_{k+1} = x_k + \alpha_k p_k$, where α_k satisfies the Wolfe conditions

$$f(x_k + \alpha_k p_k) \leqslant f(x_k) + c_1 \alpha_k p_k^T g_k \tag{3}$$
$$p_k^T \nabla f(x_k + \alpha_k p_k) \geqslant c_2 p_k^T g_k \tag{4}$$

6 Stop if the convergence criterion is met
7 **end**
8 Set $x^* = x_{k+1}$

The Newton algorithm first modifies the Hessian matrix to render it positive definite, and then calculates the direction p_k from Eq. 2 and attempts one-dimensional minimization of the objective function along line $x_k + p_k$. The complete algorithm is described in Procedure Newton.

In step 4 of Procedure Newton a direct linear algebra technique, such as Gaussian elimination or Cholesky factorization, is used to solve the Newton equation (Eq. (2)). In step 3 the Hessian matrix is first replaced by a positive definite approximation whenever this is necessary. The modification is performed by adding either a positive diagonal matrix or a full matrix to the true Hessian B_k and this can done during factorization.

In step 5 an approximate line search strategy [17] is applied to calculate the next iteration along $x_k + p_k$. This strategy searches for a suitable scalar value α_k so that two conditions of Eqs. 3 and 4 are satisfied.

The computational cost of one Newton iteration is divided in four parts: the derivative calculation, the matrix modification, the solution of the linear system and the line search. Note that, line search makes use of first order derivative information. In numerous real world optimization applications the computational cost of a single objective function evaluation, usually exceeds the cost of factorization and solution of the linear system. First and second order derivatives are even more expensive to evaluate. Hence, for most cases the computational cost of a single Newton iteration is determined by the

objective function and derivatives evaluation in steps 2 and 5 of Procedure Newton. Finally, due to its sequential nature, Procedure Newton is not an easy candidate for parallel execution.

2.2. Finite difference approximation of derivatives

It is obvious that, estimating derivatives is a common subtask in local optimization algorithms and in the solution of non-linear systems, where the Jacobian matrix is required. In a growing number of applications in science and engineering, the underlying functions are represented by large and complicated computer codes and the user may find it difficult or almost impossible to follow the original program (if it is available in source form) and develop the corresponding code for the derivative.

An alternative is offered by *finite differencing*, where the derivatives are approximated by function values at suitably chosen points. Other choices are *automatic differentiation* where the computer code for evaluating the objective function is broken down to elementary arithmetic operations and *symbolic differentiation* where algebraic expressions are generated by symbolic manipulation. 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:

$$\nabla f(x) = \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+\epsilon_i e_i) - f(x-\epsilon_i e_i)}{2\epsilon_i}, \tag{5}$$

$$\nabla^2 f(x) = \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{f(x+\epsilon_i e_i + \epsilon_j e_j) - f(x-\epsilon_i e_i + \epsilon_j e_j)}{4\epsilon_i\ \epsilon_j} - \frac{f(x+\epsilon_i e_i - \epsilon_j e_j) + f(x-\epsilon_i e_i - \epsilon_j e_j)}{4\epsilon_i\ \epsilon_j}, \tag{6}$$

where e_i is the i-th unit vector and ϵ_i, ϵ_j small positive scalars. The above formulas offer accuracy of order $O(n^2)$ and cost $2n$ and $2n^2$ function evaluations respectively.

Finite differencing is a powerful tool for approximating derivatives, which are essential in a non-linear optimization algorithm. It is also a perfect candidate for parallel execution. All function evaluations in Eq. (5), $f(x+\epsilon_i e_i)$ and $f(x+\epsilon_i e_i + \epsilon_j e_j)$, can be performed independently and in parallel. Later in this paper, we discuss specific issues and implementation details of this parallelization.

2.3. Multistart global optimization

Procedure Newton 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 very challenging, yet extremely useful, task for a wide range of scientific applications.

It can be proven, in the multidimensional case, that it is impossible to guarantee that 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 one. One of the oldest and most popular schemes for dealing 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. The strong theoretical properties of Multistart render it a widely used method. The most important extensions, that share the same principles with Multistart, are *clustering methods* [18,19], *multilevel methods* [20,21] and *random linkage methods* [22]. All the algorithms are stochastic in nature and attempt to select the best candidates for local search, whereas the simple Multistart applies local search from every sampled point.

A brief sketch of the Multistart method is presented in Procedure Multistart. Notice that the local searches (step 4) are independent and can be performed in parallel. All aforementioned variations of Multistart can benefit from a parallel execution of concurrent local searches.

Procedure. `Multistart` (f, S, x^*)

Input: Objective function, $f : S \subset \mathbb{R}^n \to \mathbb{R}$; search region: $S \subseteq \mathbb{R}^n$
Output: Approximation of the global minimizer: x^*
// Initialize the set of local minimizers
9 Set $Y = \emptyset$
10 **for** $i \leftarrow 1, 2, \ldots$ **do**
11 Sample a random point $x^{(i)}$ uniformly in S
12 Apply procedure Newton $(f, x^{(i)}, y^{(i)})$

(*continued on next page*)

(*continued*)

Procedure. `Multistart` (f, S, x^*)

```
    //Check if the local minimum is already found
13  if y^(i) ∉ Y then
14      Set Y = Y ∪ y^(i)
15  end
16  Decide whether to stop by checking an appropriate stopping rule
17 end
18 Set x* be the element of Y with the minimum function value
```

3. Multistart parallelism issues and software architecture

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 some convergence criterion is met. In each iteration, the tasks first proceed with the derivatives calculation, spawning two second-level ($L2$) tasks that compute

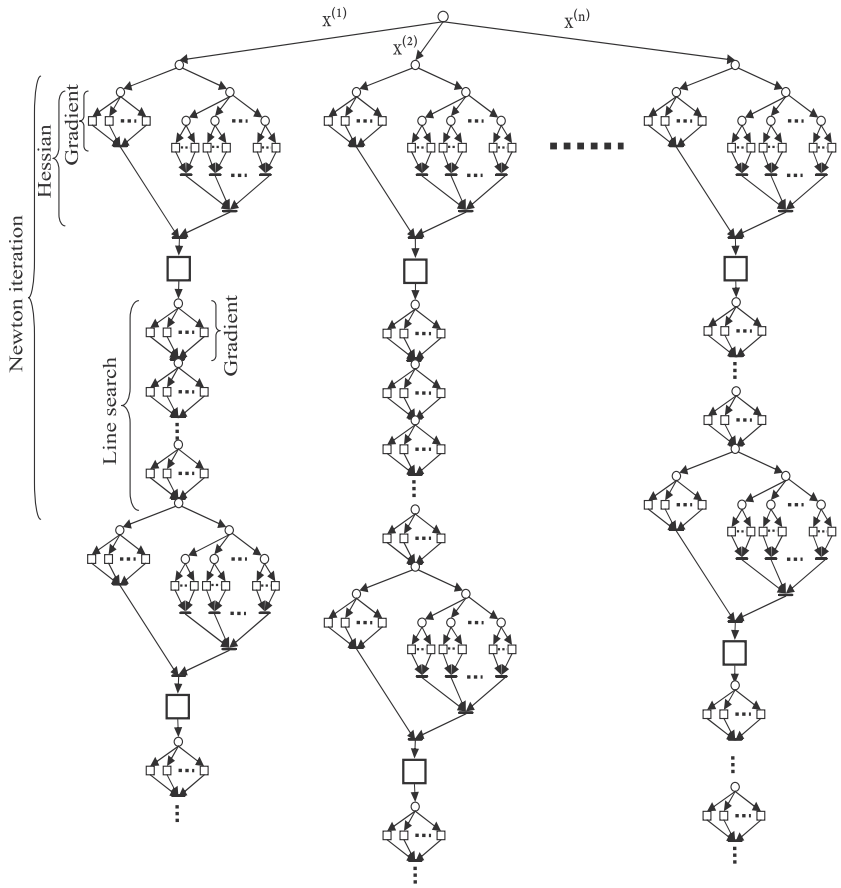

Fig. 1. Execution task graph for Multistart using finite difference derivatives.

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 2 to 9 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 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.

To deal with the significant computational demands of Multistart, a parallel implementation of the method on distributed-memory systems using MPI provides an efficient and cost-effective solution. In addition, the master-worker programming model (paradigm) can be used for handling the high irregularity of the application. According to this model, the master assigns tasks to a set of workers, sending any required input data, and then waits for the results. The workers can share some common data, which are sent by the master to them only once, but do not communicate with each other. The number of tasks usually exceeds the number of workers, and the master may generate new tasks dynamically, depending on the received results. The attractiveness of this model stems from its simplicity and inherent support for load balancing.

The master-worker model in its naive form, however, suffers from several limitations, the handling of which is not straightforward. A major drawback of the model is its low scalability, because of the bottleneck at the master that appears as the number of workers and correspondingly the number of requests that must be processed increases. Employing techniques like distributed task queues requires additional programming effort, though. Furthermore, hierarchical/nested parallelism can only be supported in a limited form, by partitioning the MPI processes into disjoint groups, one for each level of parallelism. Although an MPI-based implementation of the model can run on both multiprocessors/multicores and clusters, it always uses explicit messages and cannot easily adapt so as to take advantage of a node's physically shared memory.

All these drawbacks of the model are strongly related to the optimization problem under study in this paper. Considering the large expected number of spawned tasks (function evaluations), which can be in the order of 10^6 for a typical example, advanced runtime techniques are necessary to avoid scalability issues. Furthermore, the high irregularity makes an effective partitioning of processors impossible while the support of adaptive nested parallelism crucial. Finally, it is important to develop a hardware-independent solution, able to exploit multi-threading in a transparent way and fully take advantage of the hardware shared memory of SMP/multi-core systems.

Our system offers a parallel implementation of Multistart that deals with all the above limitations in a completely transparent way, taking advantage of two software libraries (PNDL and TORC). The architecture of our parallel optimization framework is depicted in Fig. 2 and includes the following components:

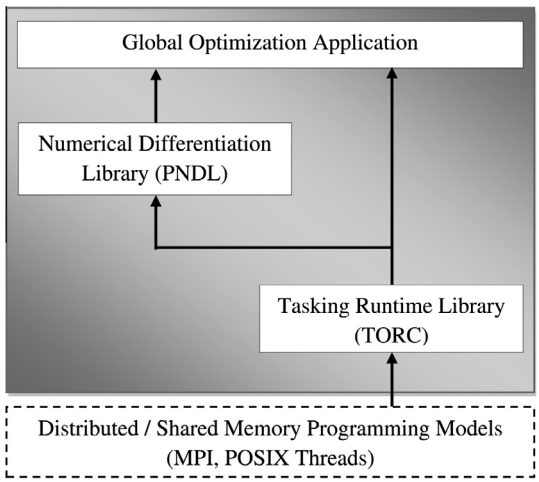

Fig. 2. Architecture of our runtime infrastructure.

- Global optimization application: It implements the parallel Multistart method taking advantage of TORC in order to spawn tasks that execute the Newton method. In addition, it issues concurrent calls to PNDL, for parallel gradient and Hessian computations.
- PNDL: It is a standalone software module that exports subroutines for calculating gradients, Hessians and Jacobians by finite differencing, supports multivariate functions, respects variable bounds and offers several prescribed accuracy levels. The parallel implementation of PNDL for multicore clusters has been based on the tasking model that TORC provides. In PNDL, task parallelism is expressed with independent function evaluations assigned to the workers.
- TORC: It is the core of our runtime environment for programming and executing irregular and adaptive master-worker applications on multi-core SMPs and clusters of such machines. TORC supports a task-style programming taking advantage of and extending the MPI programming model. TORC's is exclusively built on top of POSIX threads and MPI, offering portability, performance and a seamless integration of hardware shared-memory and message passing.

4. TORC runtime library

TORC [23] implements a task-based programming and runtime environment that make the development of master-worker applications quite straightforward. TORC assumes that a single application consists of multiple MPI processes with private memory when running on a cluster. Processes can have multiple worker threads, taking thus advantage of multiprocessor/multicore cluster nodes. Therefore, unless otherwise specified, the terms node and process are used interchangeably in this paper.

A *task* represents a work unit that is independent of its execution vehicle, i.e., the MPI process and worker thread. A spawned task can be submitted for execution to *any* MPI process; the programmer may specify the target worker in the task creation routine. 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 parent and spawn new tasks. Therefore, TORC enables the programmer to express hierarchical and recursive task parallelism naturally, which would be otherwise quite cumbersome to implement.

4.1. Programming interface

TORC provides C and Fortran interfaces for programming task-based parallel programs to be executed unaltered on both shared and distributed memory platforms. By default, TORC supports the fork-join execution model on both platforms.

Fig. 3(left) presents a complete application that uses a recursive function (`fib`) and TORC calls to compute the *N*th ($N = 50$) Fibonacci number. We observe the complete absence of explicit MPI calls and the usage of three primary TORC routines that initialize the library, spawn and join tasks.

After library initialization, performed with the `torc_init` routine, TORC transparently allows only one of the MPI processes to continue with the execution of the main routine, i.e., the primary application task, while the rest of them become workers.

TORC tasks have a parent–child relationship and can be arbitrarily nested, allowing the support of multiple levels of parallelism. In the task creation routine (`torc_task`), in addition to the target worker thread, the user must specify the task function, the number of arguments this function receives and an argument list. For each argument, its size and data type

```
#include <torc.h>

void fib (int n, unsigned long *res) {
  unsigned long res1, res2;

  if (n <= 1) {
    *res = n;
  } else {
    torc_task(-1, fib, 2,
      1, MPI_INT, CALL_BY_VAL,
      1, MPI_UNSIGNED_LONG, CALL_BY_RES,
      n-1, &res1);
    torc_task(-1, fib, 2,
      1, MPI_INT, CALL_BY_VAL,
      1, MPI_UNSIGNED_LONG, CALL_BY_RES,
      n-2, &res2);
    torc_waitall();
    *res = res1+res2;
  }
}

void main(int argc, char *argv[]) {
  unsigned long res;
  int N = 50;
  torc_init(argc, argv, MODE_MW);
  fib(N, &res);
}
```

```
#include <omp.h>

void fib (int n, unsigned long *res) {
  unsigned long res1, res2;

  if (n <= 1) {
    *res = n;
  } else {
    #pragma omp task shared(res1)
    {  fib(n-1, &res1);  }

    #pragma omp task shared(res2)
    {  fib(n-2, &res2);  }

    #pragma omp taskwait
    *res = res1+res2;
  }
}

void main(int argc, char *argv[]) {
  unsigned long res;
  int N = 50;
  #pragma omp parallel
  #pragma omp single
  fib(N, &res);
}
```

Fig. 3. Recursive Fibonacci implementation with TORC calls (left) and with OPENMP tasks (right).

is required. In addition, an intent attribute must be also supplied, similarly to the IN, OUT and INOUT intent attributes of Fortran 90. Possible values of this attribute are the following:

- CALL_BY_VAL (IN): The argument is passed by value. If the task is submitted to a remote process, a copy of the argument will be transferred to that process too.
- CALL_BY_REF (INOUT): The argument represents data that is sent along with the task and will be returned as a result in the address space of the process the task belong to.
- CALL_BY_RES (OUT): No data has to be sent but a result will be returned.

The `fib` function spawns two new tasks that are distributed across the available workers in a round-robin fashion. After task creation, a parent task calls `torc_waitall` to suspend itself until all child tasks have finished and their results are available.

The code on the right side of Fig. 3 shows the corresponding Fibonacci implementation using the OPENMP tasking model. We observe a strong similarity between the two codes. Although TORC requires some additional programming effort for the description of the task arguments, this is compensated by its support of task parallelism on distributed memory platforms.

In several applications, each task computes a partial result which is collected by the parent to produce the final result. Therefore, TORC supports *reduction operations on the results* (output arguments) of a task function. The accumulation occurs at the process of the parent task for each returned partial result. Reduction operations are supported for both scalar variables and arrays. Moreover, the reduction mechanism is multi-threaded and can be performed by multiple threads running concurrently on the same process.

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. Data broadcasting avoids unnecessary data transfers and benefits applications that otherwise would need to send the data with every task.

4.2. Implementation details

TORC has been designed to support task-based parallelism on multicores and clusters of multicores, taking advantage of and extending the MPI programming model. MPI was chosen because of its universal acceptance as the de facto messaging layer. Except for the guaranteed portability, optimized implementations of MPI are readily available for specific interconnection networks, offering a high-speed communication base.

4.2.1. Design and architecture

A TORC application consists of multiple MPI processes that run on the cluster nodes. Each process consists of a number of kernel threads that share the process memory. TORC implements a hybrid (two-level) thread architecture to support the tasking model. Each kernel thread is a worker (virtual processor) that continuously dispatches and executes ready-to-run tasks. There are private and public worker-specific and node-specific ready queues where tasks can be submitted for execution. Moreover, a server thread per process is responsible for the remote queue management and the transparent and asynchronous data movement.

Fig. 4 illustrates the general architecture of the runtime environment on a single cluster node (process).

TORC provides efficient runtime support on pure shared-memory systems too. In particular, when running on a single node, TORC uses kernel threads to exploit the multiple processors/cores and completely avoids explicit messaging. In this case, TORC is equivalent to multicore programming frameworks such as Cilk [24], Intel TBB [25] and OpenMP tasks. Considering distributed memory platforms, TORC shares some similarities with the StarPU-MPI task programming library [26]. StarPU-MPI, however, has an more complex programming interface that targets MPI clusters enhanced with GPU accelerators. TORC also supports GPU clusters [23], an extension of our optimization framework on such platforms is, however, beyond the scope of this paper.

4.2.2. Task management and data movement

A task in TORC corresponds to the remote execution of a function on a set of data that are passed as arguments to this function, in the spirit of RPC – remote procedure calls. Tasks are executed asynchronously and in any order, without any data dependencies or point-to-point communication between them.

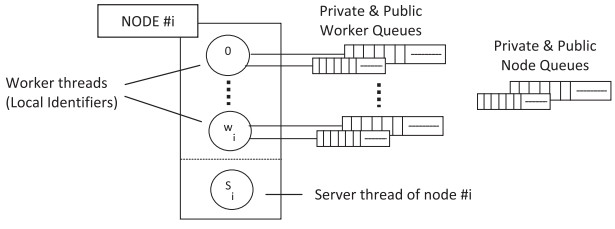

Fig. 4. Intra-node architecture of TORC.

Tasks are associated with the process (*home node*) they were created on and can be executed either locally or remotely. In the latter case, explicit movement of data takes place using MPI calls. This, however, is performed completely transparently to the user. Every data transfer relies on the communication layer of the underlying MPI implementation, which is expected to take advantage of hardware shared memory for data transfers between processes running on the same physical node.

Worker threads access the local queues through hardware shared memory and the queues of any other process through the corresponding server thread. This combination of shared memory with explicit messaging is also used for the coherence of the execution model: a task that finishes on its home node directly updates its parent. Otherwise, it sends a message to the server thread of its home node, which in turn notifies the parent task. Output results are sent back in a similar and transparent to the programmer way.

A running task that spawns parallelism can suspend its execution, waiting for the termination of its child tasks. The execution state of the task is saved, releasing the underlying kernel thread, which runs the scheduling loop for selecting the next-to-run task. When all child tasks have completed the suspended task becomes ready for execution and eventually resumes.

When the `torc_bcast` routine is called from the primary application task, which means that there are no pending tasks in the system, the blocking `MPI_Bcast` function is internally invoked. Otherwise, the worker thread that encounters an `torc_bcast` call sends asynchronously the data to the server threads of the other processes, avoiding thus any interaction with the rest of the worker threads.

4.2.3. Task distribution and scheduling

Spawning a large number of tasks can be an effective approach to distribute the work evenly among the available workers. The user can query the execution environment (number of nodes/workers) and then specify the worker where each task will be 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 (local) ready queues, visiting them in a cyclic fashion. If no work is found and inter-node task stealing is enabled, it issues requests for work to remote nodes, following a cyclic order as before. Moreover, these requests are sent synchronously, which means that a worker always waits for a response for the server thread before issuing a stealing request to the next node. A server thread that serves such a request visits the local queues in the same way as the workers.

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. The maximum number of transfers for a task and its input data cannot exceed 2; the first transfer occurs when a task is submitted for execution at a remote node, while the second one when (and if) the task is stolen by another node. As an optimization, if a task is stolen by its home node, data movement is not performed as the input task data already reside on that 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 in TORC.

5. Parallel implementation

In this section we discuss details regarding the parallelization of the numerical estimation of first and second order partial derivatives and the Multistart optimization method.

5.1. Parallel gradient and Hessian

PNDL is implemented as a Fortran software library for numerically estimating first- and second-order partial derivatives of a function by finite differencing. Various truncation schemes are offered resulting in corresponding formulas that are accurate to order $O(h)$, $O(h^2)$, and $O(h^4)$, h being the differencing step. The derivatives are calculated via forward, backward or central differences.

The implementation of PNDL for multicore clusters has been based on the tasking model that TORC provides. Task parallelism is expressed with independent function evaluations that are submitted for execution. For each function evaluation, a task is created, with main input argument a vector x and result the computed function value $f(x)$.

The core PNDL routines for gradient and Hessian computations are the following:

```
subroutine pndlg (f,x,n,iord,grad)
subroutine pndlhf (f,x,n,iord,hes)
subroutine pndlhg (g,x,n,iord,hes)
```

where f is the function to be differentiated, x is the vector containing the point of calculation, n is the dimensionality of the function, *iord* the requested order of accuracy, *grad* and *hes* are the resulting gradient vector and Hessian matrix. The `pndlhg` routine can be used if the analytical gradient g of the objective function is available. A preliminary report on the intended PNDL API is available in [27]. In contrast to [27], however, the parallel routines that PNDL exports to MPI programs have been

redesigned for a master-worker, instead of SPMD execution mode. As such, the calling process initializes the input parameters and receives the computed derivatives.

Fig. 5 illustrates the way PNDL uses TORC calls to calculate the gradient, exploiting a single level of parallelism, as depicted in Fig. 1. In particular, when `pndlg` is invoked, the primary task initially broadcasts the input vector x through the use of a common block. It then spawns and distributes cyclically function evaluation tasks to the workers. After task completion, the primary task first gathers all the required function values and then computes the derivatives according to the given numerical differentiation formula. An alternative is to have each task compute a part of the numerical formula and then combine its result through a reduction operation. The advantage, however, of the adopted two-phase scheme (gathering of function values and then estimation of the derivative) is that it preserves the sequential order of calculations and, thus, avoids rounding errors due to reordering.

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 s-level tasks, according to user parameters (selected numerical differentiation formula, desired accuracy and bounds).

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. 6 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 (`istridel`) 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. Fig. 7 illustrates an example of the STRIDE distribution scheme: each second-level task corresponds to a single function call. The number inside each task denotes which of the 8 workers will be assigned that task.

Although the `STRIDE` scheme balances the number of function evaluations betweens processors, it is 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 (`istridel = 1`) and submits the second-level tasks locally (`istride2 = 0`) with task stealing enabled. An example of the GLTS scheme is depicted in Fig. 8. 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 configuration of each distribution scheme is summarized in Table 1.

Despite the availability of several software packages for estimating derivatives numerically (e.g., [28–30]) their implementation is sequential. In [31], the authors present an MPI-based parallel numerical Hessian implementation. They use the central difference formula and follow a sequential or block, though static, decomposition method for distributing function evaluations to processors. In contrast, PNDL uses a dynamic two-level task distribution scheme which is further enhanced with task stealing. Our approach allows for multiple concurrent Hessian calculations and better load balancing.

5.2. Parallel Newton and Multistart methods

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 common blocks for broadcasting the input vector. Therefore, we use an array of input vectors in the common block and each PNDL function call is dynamically

```
! first level
subroutine pndlg(f, x, n, iord, grad)
external f
integer n, iord
double precision x(n), grad(n), xx(n)
...
<set xx(I) = x(I)>  ! create copy of input vector x
call torc_bcast(xx, n, MPI_DOUBLE_PRECISION) ! broadcast copy
<for each required function value>
  call torc_task(-1, f, ..) ! cyclic distribution
call torc_waitall()
<compute vector grad>
end
```

Fig. 5. Outline of a PNDL gradient calculation with exploitation of a single level of parallelism.

```
! first level
subroutine pndlhf(f, x, n, iord, hes)
external f, driver
integer n, iord
double precision x(n), hes(n,n), xx(n)
...
<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, ...)
...
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. 6. Outline of a PNDL Hessian calculation with exploitation of two levels of parallelism using the STRIDE distribution scheme.

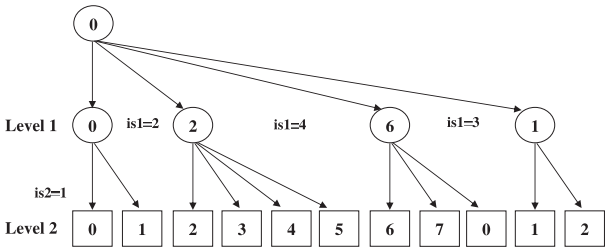

Fig. 7. An example of the STRIDE task distribution scheme for the estimation of second order derivatives ($is \equiv istride$).

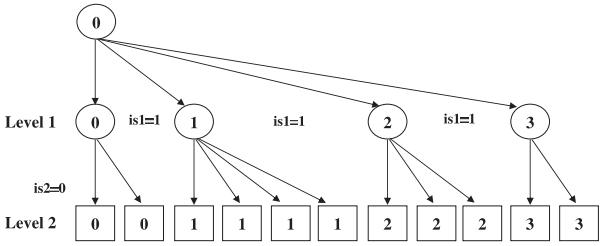

Fig. 8. An example of the GLTS task distribution scheme.

Table 1
Configuration of the various task distribution schemes.

Distribution scheme	istride1	istride2	Internode stealing
STRIDE	variable	1	no
GL	1	0	no
GLTS	1	0	yes
LLTS	0	0	yes

assigned a unique identifier that specifies an available entry of this array. The size of the array denotes the maximum number of concurrent PNDL functions that can be simultaneously active.

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 with inter-node 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. For a single Newton-based local search, two entries in the global array are sufficient. Therefore, for N local searches the number of entries in the global array can be $2P$ instead of $2N$, where P is the maximum number of workers.

6. Protein conformation

The protein conformation problem is defined in [32] as the problem of determining the three dimensional structure of a protein, called its *tertiary structure*, just from the sequence of amino acids that it is composed of (its *primary structure*). Under the assumption that in the native state the potential energy of a protein is globally minimized, the protein conformation problem can be regarded as equivalent to solving the problem

$$\min_{x\in\mathbb{R}^{3n}} E(x) \tag{7}$$

where $E(x)$ is the value of a potential energy function for an n atom protein described by a $3n$ dimensional coordinate vector. This optimization problem has a large number of variables, depending on the size of the amino acid sequence, and many local minimizers that grow exponentially with the number of atoms.

Protein folding is considered one of the most challenging global optimization problems due to the vast number of local optimal conformations and the large objective function computation time. Until recently, only small protein structures were examined thoroughly and their global minimum conformations were revealed.

To model the potential energy of Eq. 7 we used the Tinker [16] software, a flexible system of programs and routines for molecular mechanics and dynamics. The modeled energy depends on the positions of the atoms in 3D-space and includes distance and angle calculations. In our application we minimize a reduced scheme of dihedral angles between consecutive triads of atoms.

7. Results and discussion

We have experimented with all the components of our software system extensively. In this section we present performance results on a dedicated Sun Fire ×4100 cluster of 16 nodes interconnected with Gigabit Ethernet. Each node has 2 dual core AMD Opteron-275 processors running at 2.2 GHz for a total of 64 cores. The software was compiled under Linux 2.6 with the GNU gcc compiler and MPICH2 [33]. The multithreaded-safe MPI implementation allows for blocking receive calls in the loop of the server thread that avoid polling and thus minimize any interference with the worker threads of the application.

Thanks to the design of TORC, the same application binary can exploit the 4 processor cores of a single node with several combinations in the number of processes and workers. For example, the application may have 4 processes of one worker each or a single process with 4 worker threads.

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 for a zero-argument task is measured approximately 0.1 ms and depends on the latency of the interconnection network. In contrast, within a multi-core node we support very fine-grained tasks efficiently.

7.1. PNDL-Parallel Hessian computation

We present two sets of synthetic experiments that calculate the Hessian with O (h^4) precision without imposing bounds on the variables. We used as benchmark the Rastrigin function

$$f(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i)),$$

with artificial delay. We consider only the Hessian, since the parallel Gradient calculation is a straightforward case of cyclic task distribution.

The first set of experiments (E1 and 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 100 ms and 1000 ms via appropriate artificial delays. The second set (E3 and E4) uses a 100-dimensional test function with artificial delays of 10 ms and 100 ms. The number of function evaluations for this set is 20100. 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 (the function value may be the result of a simulation).

Fig. 9(a)–(d) present the results from the two sets of experiments with the 4 task distribution schemes (STRIDE, GLTS, GL, and LLTS). For the first set of experiments (Fig. 9(a) and (b)), 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 a small serial fraction of code in the PNDL routine and the load imbalance when the number of function evaluations is not exactly divided by the number of workers. The overhead of broadcasting the 160 bytes of the input point is negligible and does not affect the overall performance of the application. Although all distribution schemes exhibit comparable performance up to 32 processors, GLTS achieves the highest speedup on 64 processors, with LLTS and GL following. The lowest speedup corresponds to the STRIDE scheme because of its large number of explicit messages.

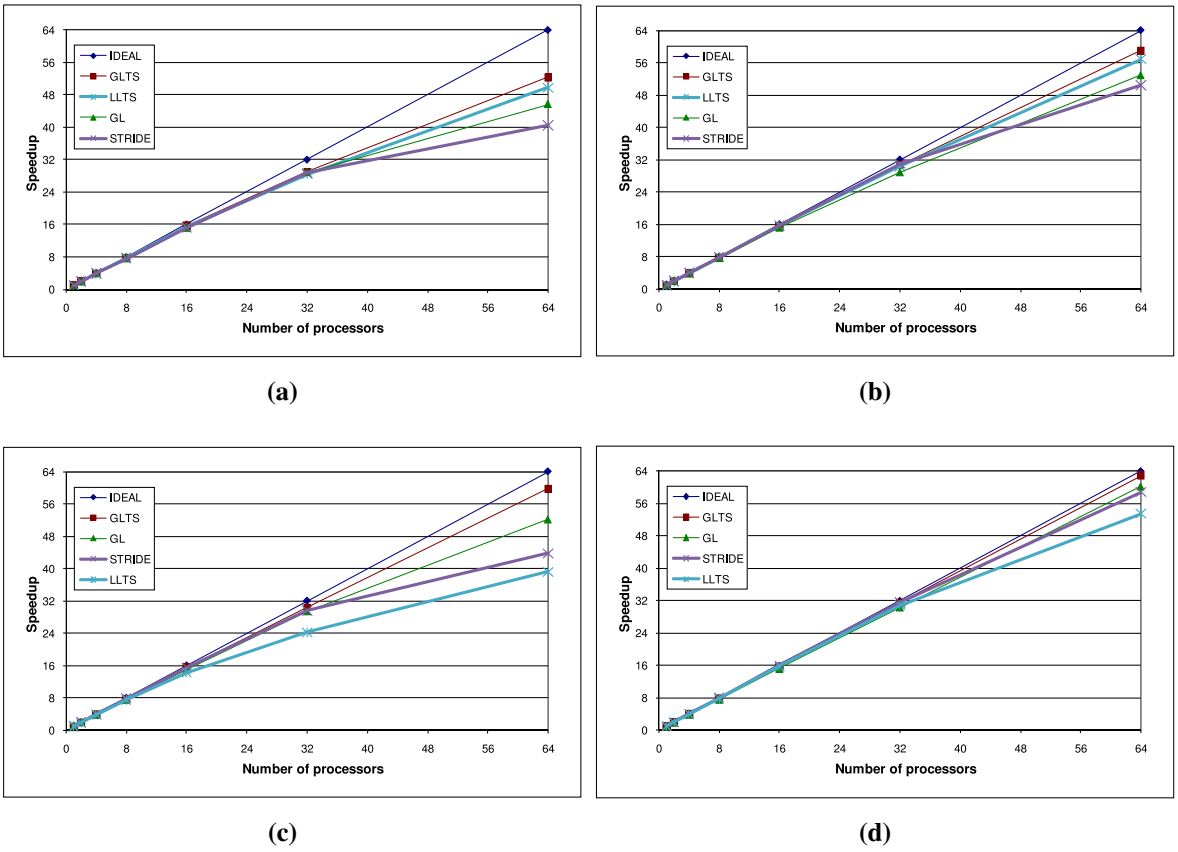

Fig. 9. Speedup measurements. (a) E1 (variables = 20, delay = 100 ms). (b) E2 (variables = 20, delay = 1000 ms). (c) E3 (variables = 100, delay = 10 ms). (d) E4 (variables = 100, delay = 100 ms).

For the 100-dimensional test function (Fig. 9(c) and (d)), 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.

7.2. Multistart-parallel global optimization

In order to evaluate parallel Multistart, we also used the Rastrigin test function with 10 variables and artificial delays that range from 1 ms to 1000 ms. We provide results for $1, 16, 64$ and 1024 initial points, using the modified LLTS (MLTS) task distribution scheme in all cases.

Fig. 10(a) 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 a small serial fraction ($\simeq$2%) of the Newton code and specifically to the direct linear algebra part that includes a Cholesky factorization that has not been parallelized. 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, we observe that the measured speedup is very close to the maximum theoretical one. The latter, according to Amdahl's law [34], is defined as

$$S(p) = \frac{1}{b + (1-b)/p},$$

where p is the number of processors and b the fraction of the code which is strictly serial (2% in our case).

Fig. 10(b)–(d) show the speedup of Multistart when 16, 64 and 1024 optimizations are issued. The attained 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 10 ms and 100 ms function evaluation time.

The next experiment studies the performance behavior of Multistart with respect to the number of variables. Fig. 11(a) and (b) depict the speedup on 64 workers for 64 initial points and function evaluation time equal to 1 ms and 10 ms

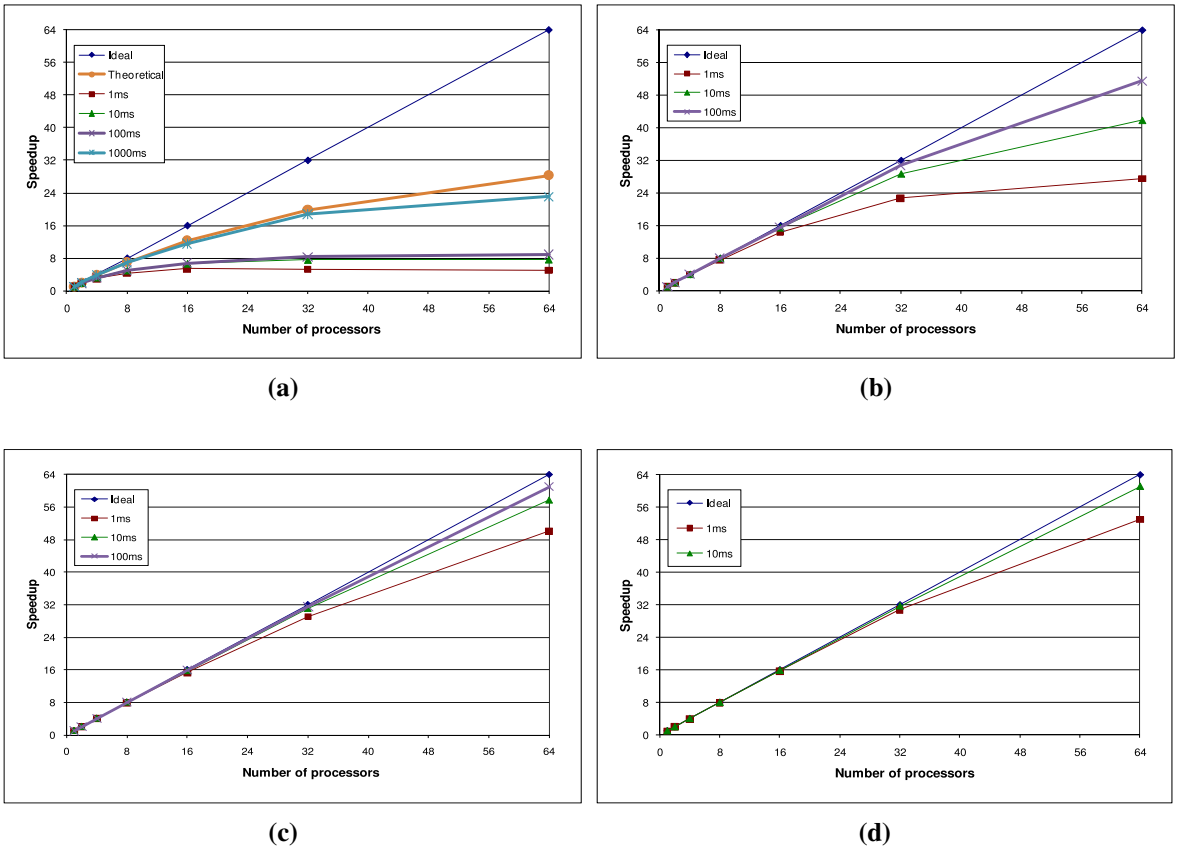

Fig. 10. Speedup for varying number of local searches. (a) 1 local search. (b) 16 local search. (c) 64 local search. (d) 1024 local search.

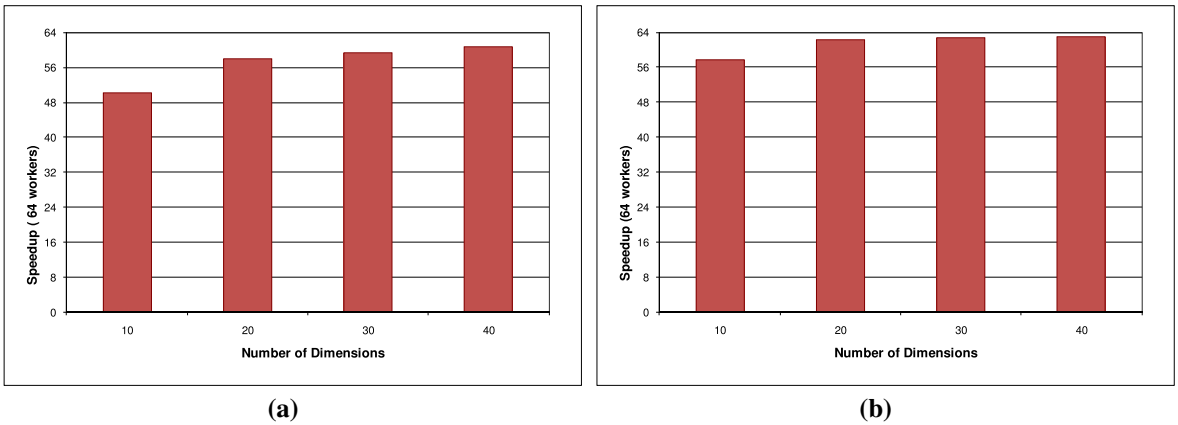

Fig. 11. Speedup on 64 workers for 64 local searches for 10, 20, 30 and 40 variables. (a) 1 ms function evaluation time. (b) 10 ms function evaluation time.

respectively. We observe that the obtained speedup increases with the number of objective function variables. For 10 ms delay, the obtained speedup is significantly higher for 10 variables and approximates the ideal if more function variables are used.

7.3. Multistart-protein conformation

The protein tested consists of 8 Alaline amino acids (Polyalaline-8) in the primary structure chain, blocked by the ACE and NME groups. The resulting global optimization problem consists of 35 parameters in internal coordinate space. The goal is to reproduce the already known global minimum conformation of Polyalaline-8 that was first explored in [35].

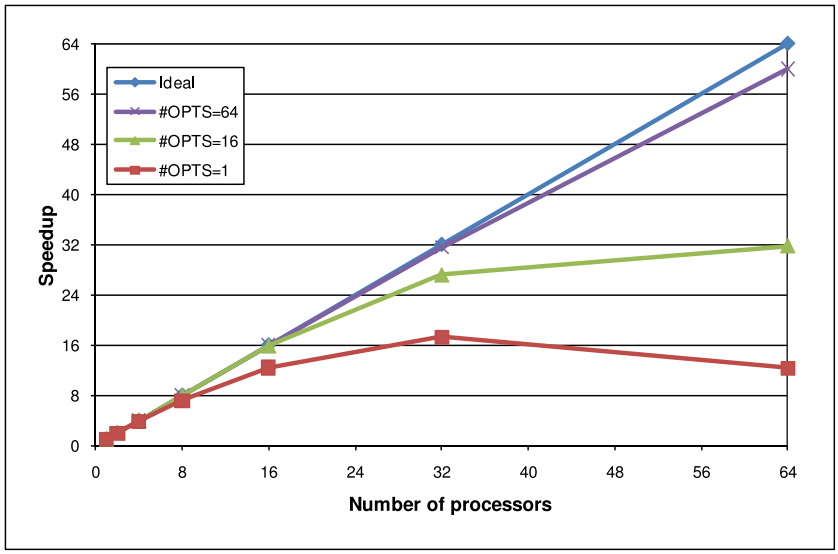

Fig. 12. Speedup for 1, 16 and 64 local searches for Polyalaline-8.

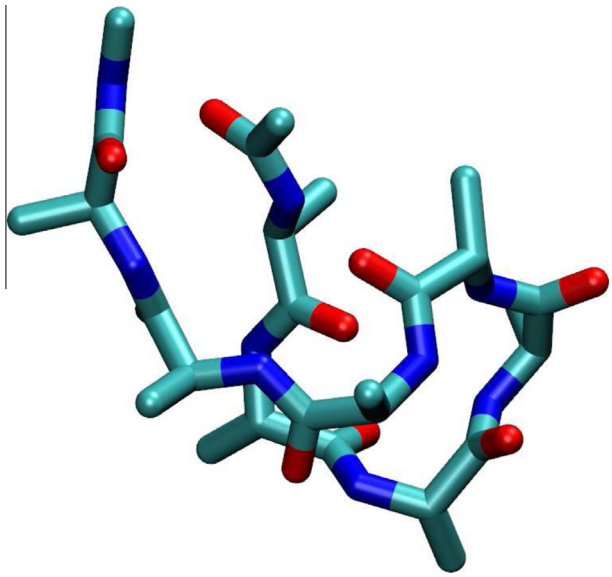

Fig. 13. Polyalaline-8 minimum energy conformation, $E_{AMBER96} = -44.45$ kcal/mol.

Fig. 12 shows the speedup of Multistart for 1, 16 and 64 initial points. Every energy function performed by Tinker takes approximately 4 ms for Polyalaline-8. We observe that the speedup increases with the number of independent optimizations and that the results match those presented previously using the test function. As the number of initial points increases, the speedup of Multistart approaches the ideal as better load balancing is achieved.

In all the experiments we compute the Hessian matrix numerically using function evaluations; an alternative approach is to use the analytic gradient routine of the objective function, if this is available. This reduces the number of function evaluations of Multistart and equivalently the number of spawned tasks. For instance, the computational cost of the gradient function for Polyalaline-8 is approximately 1 ms. When gradient calls are used, the performance scalability of Multistart remains the same, the only difference being the considerably smaller execution time of each local optimization.

In Fig. 13 we present the final conformation of Polyalaline-8 as computed by the parallel Multistart. The result matches the global best conformation presented in [35].

8. Conclusions

We presented the design and implementation details of an optimization framework which efficiently exploits nested and irregular parallelism on clusters of multicores/SMPs. At the core of our framework is TORC, a runtime library that supports dynamic task-based parallelism on these platforms. 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. The scalability of our system was demonstrated on a multicore cluster with synthetic benchmarks and validated with a real application case that deals with the protein conformation problem. The experiments results showed that our system achieves speedups that coincide with the ideal one, verifying that TORC manages to handle efficiently the irregular task parallelism of Multistart. Going a step further we believe that the runtime environment presented can support much more complicated global optimization schemes.

Our future plans include the integration of additional numerical optimization techniques into our infrastructure. Furthermore, we currently extend the applicability of our system to computational grids and heterogeneous environments.

References

[1] J.F. Schutte, J.A. Reinbolt, B.J. Fregly, R.T. Haftka, A.D. George, Parallel global optimization with the particle swarm algorithm, Int. J. Numer. Methods Eng. 61 (13) (2004) 2296–2315.
[2] J. He, L.T. Watson, M. Sosonkina, Algorithm 897: VTDIRECT95: serial and parallel codes for the global optimization algorithm DIRECT, ACM Trans. Math. Soft. (TOMS) 36 (3) (2009) 17:1–17:24.
[3] R.H. Byrd, E. Eskow, A. van der Hoek, R.B. Schnabel, K.P.B. Oldenkamp, A parallel global optimization method for solving molecular cluster and polymer conformation problems, 7th Siam Conf. on Parallel Processing for Scientific Comput., SIAM, Philadelphia, 1995, pp. 72–77.
[4] R.H. Byrd, E. Eskow, A. van Der Hoek, R.B. Schnabel, C.S. Shao, Z. Zou, Global optimization methods for protein folding problems, Global minimization of nonconvex energy functions: molecular conformation and protein folding, Am. Math. Soc. (1996) 29–39.
[5] T.F. Coleman, Z. Wu, Parallel continuation-based global optimization for molecular conformation and protein folding, J. Global Optim. 8 (1) (1996) 49–65.
[6] S. Crivelli, T. Head-Gordon, R. Byrd, E. Eskow, R. Schnabel, A hierarchical approach for parallelization of a global optimization method for protein structure prediction, in: 5th Int'l Euro-Par Conf. on Parallel Processing, Toulouse, France, 1999, pp. 579–585.
[7] S. Kozola, Improving optimization performance with parallel computing, The MathWorks Inc., Tech. Rep. 91710v00 (2009). pp. 1–6. Available at http://www.mathworks.ch/company/newsletters/articles/improving-optimization-performance-with-parallel-computing.html.
[8] J. Eriksson, P. Lindstrom, A parallel interval method implementation for global optimization using dynamic load balancing, Reliable Comput. 1 (2) (1995) 77–91.
[9] C. Hu, B. Kearfott, S. Xu, X. Yang, A parallel software package for nonlinear global optimization, in: 5th Int'l Conf. on Optimization: Techniques and Applications, Hong Kong, 2001.
[10] K. He, L. Zheng, S. Dong, L. Tang, J. Wu, C. Zheng, PGO: a parallel computing platform for global optimization based on genetic algorithm, Comput. Geosci. 33 (3) (2006) 357–366.
[11] M. Wahib, M. Munetomo, A. Munawar, K. Akama, 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 (PDCAT' 07), Washington, DC, 2007, pp. 167–168.
[12] A. Günay, F. Öztoprak, Ş. Birbil, P. Yolum, Solving global optimization problems using MANGO, in: 3rd KES Int'l Symp. on Agent and Multi-Agent Systems: Technologies and Applic., Upsalla, Sweden, 2009, pp. 783–792.
[13] F. Biscani, D. Izzo, C. Yam, A global optimisation toolbox for massively parallel engineering optimisation, in: 4th Int'l Conf. on Astrodynamics Tools and Techniques, Madrid, Spain, 2010.
[14] J. He, M. Sosonkina, C.A. Shaffer, J.J. Tyson, L.T. Watson, J.W. Zwolak, A hierarchical parallel scheme for a global search algorithm, High Performance Computing Symposium, Advanced Simulation Technologies Conference, International Society for Modeling and Simulation, 2004, pp. 43–50.
[15] A.H.G. Rinnooy Kan, C.G.E. Boender, Bayesian stopping rules for multistart global optimization methods, Math. Program. 37 (1) (1987) 59–80.
[16] J.W. Ponder, TINKER-Software Tools for Molecular Design, Washington University, St. Louis, 1999. Version 3.7.
[17] J.J. Moré, D.J. Thuente, Line search algorithms with guaranteed sufficient decrease, ACM Trans. Math. Soft. (TOMS) 20 (3) (1994) 286–307.
[18] A.H.G. Rinnooy Kan, G.T. Timmer, Stochastic global optimization methods Part I: clustering methods, Math. Program. 39 (1987) 27–56.
[19] C. Voglis, I.E. Lagaris, Towards ideal multistart, a stochastic approach for locating the minima of a continuous function inside a bounded domain, Appl. Math. Comput. 213 (2009) 216–229.
[20] A.H.G. Rinnooy Kan, G.T. Timmer, Stochastic global optimization methods Part I: clustering methods, Math. Program. 39 (1987) 57–78.
[21] M.M. Ali, C. Storey, Topographical multilevel single linkage, J. Global Optim. 5 (1994) 349–358.
[22] M. Locatelli, F. Schoen, Random Linkage: a family of acceptance/rejection algorithms for global optimisation, Math. Program. 85 (2) (1999) 379–396.
[23] P.E. Hadjidoukas, E. Lappas, V.V. Dimakopoulos, A tasking library for platform-independent task parallelism, in: 20th Euromicro Int'l Conf. on Parallel, Distributed and Network-Based Processing, Munich, 2009, pp. 229–236.
[24] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, J. Parallel Distrib. Comput. 37 (1) (1996) 55–69.
[25] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism, O' Reilly Media Inc, 2007.
[26] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, S. Thibault, StarPU-MPI: task programming over clusters of machines enhanced with accelerators, in: 19th European MPI Users' Group Meeting, Vienna, Austria, 2012.
[27] C. Voglis, P.E. Hadjidoukas, I.E. Lagaris, D.G. Papageorgiou, A numerical differentiation library exploiting parallel architectures, Comput. Phys. Commun. 180 (8) (2009) 1404–1415.
[28] GSL, Gnu scientific library. Available at <http://www.gnu.org/software/gsl/>, 2012.
[29] NAG Fortran Library, D04 numerical differentiation, subroutine D04AAF.
[30] P. Gilbert, R. Varadhan, numDeriv: accurate numerical derivatives. Available at <http://cran.r-project.org/web/packages/numDeriv/>, 2012.
[31] M.S. Staveley, R.A. Poirier, S.D. Bungay, An evaluation of parallel numerical Hessian calculations, in: High Perf. Comput. Symp. (HPCS 2009), LNCS, vol. 5976, 2009, pp. 196–214.
[32] C.B. Anfinsen, Principles that govern the folding of protein chains, Science 181 (1973) 223–230.
[33] W. Gropp, mpich2: a new start for mpiimplementations, in: 9th European PVM/MPI Users' Group Meeting, Linz, Austria, 2002, pp. 37–42.
[34] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: The Spring Joint Computer Conference, April 18–20, 1967, ACM, 1967, pp. 483–485.
[35] P.N. Mortenson, D.J. Wales, Energy landscapes, global optimization and dynamics of the polyalanine Ac (ala) NHMe, J. Chem. Phys. 114 (2001) 6443–6454.