PROCEEDINGS OF
THE 2006 INTERNATIONAL CONFERENCE ON SCIENTIFIC COMPUTING

# CSC'06

**Editor**

**Hamid R. Arabnia**

**Associate Editors**

**George A. Gravvanis**
**Ashu M. G. Solo**

Las Vegas, Nevada, USA
June 26-29, 2006

# Matrix Distributed Processing and Applicaitons

Massimo Di Pierro
School of Computer Science, Telecommunications and Information Systems
DePaul University, 243 S. Wabash Av., Chicago, IL 60604, USA

April 30, 2006

**Abstract**

Matrix Distributed Processing (MDP) is a C++ library for fast development of efficient parallel algorithms. MDP is based on MPI and consists of a collection of C++ classes and functions such as lattice, site and field. Algorithms using these components are automatically parallel and no explicit call to communication functions is required. MDP is particularly suitable for implementing parallel solvers for multi-dimensional differential equations and mesh-like problems. MDP includes a simulator that allows one to run and test parallel programs on a single node.

## 1 Introduction

Matrix Distributed Processing (MDP) [1] is a collection of classes and functions written in C++ for fast development of parallel algorithms such as solvers for partial differential equations, mesh-like algorithms, and various types of graph-based problems. These algorithms find frequent application in many sectors of physics, engineering, electronics and computational finance. MDP was originally developed to simplify the coding of parallel Lattice QCD algorithms, i.e. the numerical computation of properties of composite particles made of quarks, such as protons and neutrons. While Lattice QCD is currently one of the main applications of MDP, its range of applicability is not limited to it.

MDP is based on Message Passing Interface (MPI) but parallelization is transparent to the programmer, who does not need to use explicit calls to send/receive functions. It includes functions for linear algebra with support of a Maple-like syntax, statistical functions, and fitting functions. MDP also includes a Parallel SIMulator (PSIM) so that algorithms developed using MDP can be run in parallel on a single processor machine without MPI. The parallel processes are created by forking and communications are realized using local sockets.

MDP can be used on any machine with ANSI C++, POSIX, and MPI. No specific communication hardware is required, but a fast network switch is suggested.

The best way to introduce MDP is by example. Here is an example of how to solve a 3d Laplace equation recursively using MDP.

**Problem:** Consider the following Laplace equation:

$$\nabla^2\varphi(x) = f(x) \tag{1}$$

where $\varphi(x)$ is a field of $2 \times 2$ complex matrices defined on a 3D space (space), $x = (x_0, x_1, x_2)$ limited by $0 \leq x_i < L_i$, and

$$\begin{aligned} L &= \{10, 10, 10\}, \\ f(x) &= A\sin(2\pi x_1/L_1), \\ A &= \begin{pmatrix} 1 & i \\ 3 & 1 \end{pmatrix} \end{aligned} \tag{2}$$

The initial conditions are $\varphi_{initial}(x) = 0$. We will also assume that $x_i + L_i = x_i$ (torus topology).

**Solution:** In order to solve eq. (1) we first discretize the Laplacian ($\nabla^2 = \partial_0^2 + \partial_1^2 + \partial_2^2$) and rewrite it as

$$\sum_{\mu=0,1,2} [\varphi(x + \widehat{\mu}) - 2\varphi(x) + \varphi(x - \widehat{\mu})] = f(x) \tag{3}$$

where $\widehat{\mu}$ is a unit vector in the discretized space in direction $\mu$. Hence we solve it in $\varphi(x)$ and obtain the following a recurrence relation

$$\varphi(x) = \frac{\sum_{\mu=0,1,2} [\varphi(x + \widehat{\mu}) + \varphi(x - \widehat{\mu})] - f(x)}{6} \tag{4}$$

The following is a typical MDP program that solves eq. (1) by recursively iterating eq. (4). Notice how the program is parallel but there are no explicit call to communication functions:

```
00  #include "mdp.h"
01
02  void main(int argc, char** argv) {
03     mdp.open_wormholes(argc,argv);     // open communications
04     int L[]={10,10,10};                // declare volume
05     mdp_lattice      space(3,L);       // declare lattice
06     mdp_site         x(space);         // declare site variable
07     mdp_matrix_field phi(space,2,2);   // declare field of 2x2
08     mdp_matrix       A(2,2);           // declare matrix A
09     A(0,0)=1;  A(0,1)=I;
10     A(1,0)=3;  A(1,1)=1;
11     forallsites(x)                     // loop (in parallel)
12        phi(x)=0;                       // initialize the field
13     phi.update();                      // communicate!
14
15     for(int i=0; i<1000; i++) {        // iterate 1000 times
16        forallsites(x)                  // loop (in parallel)
```

```
17          phi(x)=(phi(x+0)+phi(x-0)+
18                  phi(x+1)+phi(x-1)+
19                  phi(x+2)+phi(x-2)-
20                  A*sin(2.0*Pi*x(1)/L[1]))/6;  // equation
21       phi.update();                  // communicate!
22    }
23    phi.save(''field_phi.mdp'');        // save field
24    mdp.close_wormholes();             // close communications
25 }
```

Notes:

- Line 00 includes the MDP library.
- Lines 03 and 25 respectively open and close the communication channels over the parallel processes.
- Line 04 declares the size of the box $L = \{L_0, L_1, L_2\}$.
- Line 05 declares a 3-dimensional lattice, called `space`, on the box $L$. MDP supports up to 10-dimensional lattices. By default, a lattice object is a mesh with torus topology. It is possible to specify alternate topologies, boundary conditions, or parallel partitioning for the lattice. Note that each lattice object contains a parallel random generator.
- Line 06 declares a variable site, called `x`, that will be used to loop over lattice sites, in parallel.
- Line 07 declares a field of $2 \times 2$ matrices, called `phi`, over the lattice `space`. MDP is not limited to fields of matrices. It is easy to declare fields of any user-defined structure or class.
- Lines 08 through 10 define the matrix A.
- Lines 11 and 12 initialize the field `phi`. Notice that `phi` is distributed over the parallel processes and `forallsites` is a parallel loop.
- Line 13 performs communications so that each process becomes aware of changes in the field performed by other processes (*synchronization*).
- Lines 15 through 23 perform 1000 iterations to guarantee convergence. In real life applications one may want to implement some convergence criteria as stopping condition.
- Line 16 loops over all sites in parallel.
- Lines 17 through 20 implement eq. (4). Notice the similarity in notation. Here `phi(x)` is a $2 \times 2$ complex matrix.
- Line 21 performs *synchronization.*

```
00  #include "mdp.h"
01  void main(int argc, char** argv) {
02    mdp.open_wormholes(argc,argv);
03    int L[]={100};
04    mdp_lattice line(1,L);
05    mdp_field<int> spin(line);
06    mdp_site x(line);
07    int dE=0, H=L[0], dH=0;
08    float kappa=2.0;
09    forallsites(x) spin(x)=+1;
10    while(1) {
11      dH=0;
12      for(int parity=EVEN; parity<=ODD; parity++) {
13        forallsitesofparity(x,parity) {
14          dE=2*spin(x)*(spin(x-0)+spin(x+0));
15          if(exp(-kappa*dE)>mdp_random.plain())
16            { spin(x)*=-1; dH=dH+2*spin(x); }
17        }
18        spin.update(parity);
19      }
20      mdp.add(dH);
21      H=H+dH;
22      mdp << ''magnetization='' << H << endl;
23    }
24    mdp.close_wormholes();
25  }
```

In this example lines 3-4 declare a 1D lattice of 100 points (`line`). Line 5 declares a field of integers (`spin`) on this lattice. Line 9 sets all field variables to 1 and line 7 sets the total magnetization H for this initial spin configuartion.

Line 13 computes the energy variation (`dE`) of each Markov Chain Monte Carlo (MCMC) step. Lines 15-16 perform the Monte Carlo accept-reject. If a change is accepted the spin at site x is flipped and the total magnetization changes (line 16).

Note how at each MCMC step, first the code tries to flip the spins at even locations then, after it updates the lattice sites, it tries to flip the spins at odd locations (line 13). This guarantees computation results are independent on parallelization of the lattice line.

Since this even-odd distinction is common in many lattice algorithms, `MDP` stores all even lattice sites and all odd lattice sites close together. This speeds up loops over one of the two subsets and also speeds up communication. In fact, in this example, we are able to limit the synchronization (update) to the site of a given parity (line 18).

- Line 23 saves the field. Note that all fields, including user-defined ones, inherit `save` and `load` methods from the basic `mdp_field` class.
- It should also be noted that all MDP classes and functions are both type and exception safe. Moreover MDP components can be used without knowledge of C pointers and pointer arithmetics.

## 2 Linear Algebra and Other Tools

MDP includes a Linear Algebra package and other tools. Some of the most important classes are:

- class `mdp_real`, that should be used in place of float or double;
- class `mdp_complex`, for complex numbers;
- class `mdp_array`, for vectors and/or multidimensional tensors;
- class `mdp_matrix`, for any kind of complex rectangular matrix;
- class `mdp_measure`, for error propagation;
- class `mdp_jackboot`, a container for jackknife and bootstrap algorithms.

The most notably difference between our linear algebra package and other existing packages is its natural syntax.

For example:

```
mdp_matrix A,B;
A=Random.SU(7);
B=exp(A)+inv(A)*hermitian(A)+5;
```

reads like

$A$ and $B$ are matrices

$A$ is a random $SU(7)$ matrix

$B = e^A + A^{-1}A^H + 5 \cdot \mathbf{1}$

Note that each matrix can be resized at will and is resized automatically when a value is assigned.

MDP includes functions for fitting such as the Levenberger-Marquardt algorithm.

## 3 Lattice, Site, and Field

An `mdp_lattice` is the class that describes the space on which fields are defined; it stores the *topology* of the space (by default that of a torus in $d$ dimensions) and information about *partioning* of the space over the parallel processes. In MDP, a lattice is a graph, defined as a collection of points (lattice *sites*) connected

by links (they specify the topology). Each site is uniquely mapped to one of the parallel processes.

The only restriction is that the graph must have a degree less than 20. From now on we will assume the default topology of a torus, therefore the lattice should be thought of as a mesh in $d \leq 10$ dimensions.

The constructor class `mdp_lattice` determines on which process to store each site, determines the neighbors of each site, and the sizes of the buffers where each process keeps copies of those sites that are non-local but are neighbors of the local sites.

The constructor also allocates a parallel random number generator so that each site of the lattice has its own independent random number generator. This is important for parallel Monte Carlo applications of `MDP` and ensures reproducibility of computations on different architectures.

On each lattice it is possible to allocate fields. Some fields are built-in, for example `mdp_field<mdp_complex>`, i.e. the field of complex numbers. The user can declare any type of field. For example a field of 5 float per lattice site:

```
class S {
public: float S[5];
};
int L[]={10,10,10};
mdp_lattice cube(3,L);
mdp_field<S> psi(cube);
```

This code declares a $10 \times 10 \times 10$ lattice (`cube`) and a field (`psi`), that lives on the `cube`. The site variables of `psi`, `psi(x)` belong to class `S` (assuming `x` is an `mdp_site` on the `cube`).

User-defined fields can be saved:

```
psi.save("filename");
```

loaded

```
psi.load("filename");
```

and synchronized

```
psi.update();
```

Synchronization means that all processes will perform MPI communications to make sure all buffers that contain copies of non-local site variables are updated with the proper values. The method update should be called immediately after the local site variables of a field have been changed.

Once a field object is declared, in the field constructor, each process dynamically allocates memory for the buffers that store the copies of those sites that are non-local but are neighbors of the local sites. These buffers are created in such as way to ensure optimal communication patterns.

Every time a field changes, for example in a parallel loop such as

```
forallsites(x) phi(x)=0;
```

the program notifies the field that its values have been changed by calling

```
phi.update();
```

The method `update` performs all required communication to copy site variables that need to be synchronized between each couple of overlapping processes.

Lattice sites are represented by objects of class `mdp_site`. Site objects can be looped over in parallel loops (such as forallsites) but it also possible to explicitly address a specific site by specifying the site coordinates. Obviously only the process that stores a site locally should address a specific site. Class `mdp_site` has methods to check if a site is local, is it is non-local but a local copy is present and which process stores the site locally.

# 4 Optimal Communication Patterns

In MDP, the lattice objects, according with the lattice topology and the parallel partitioning, determines the optimal way to store site variables in memory and performing parallel communication. This information is then used by the field method update that performs the synchronization of the field variables.

Note that MDP does not attempt to overlap computation and communication. By optimal communication pattern we mean that, under the assumptions below, the method update minimize network traffic and data copies.

Current optimizations are based on the assumption that each processing node has one and only one network card at the same time and that the network is isotropic (latency and bandwidth for each couple of nodes is the same). This assumption is generally true for Ethernet and Myrinet clusters.

Communications are optimal in the sense that:

- Each process retrieves all non-local site variables in a single send/recv for each process that contains sites which are neighbors of local sites.
- Two processes that do not store neighbor sites do not communicate with each other.
- No process is involved in more than a single send and a single receive at one time.
- Each process stores close in memory those copies of non-local site variables which are local to the same process. In this way synchronization does not require the use of additional buffers receiving buffers.

Note that two processes may be overlapping (i.e., store neighbor sites) with respect to one lattice and not overlapping in respect to a different lattice within the same program. It is possible, in principle, to change the above communication patterns to optimize communication for various network topologies.

Although communication is currently based on MPI, they do not make use of communication tags. It is therefore possible, in principle, to speed-up communication by using a faster tagless and bufferless protocol such as Myricom GM.

Our communication patterns have the effect of making communication almost insensitive to network latency, and communication speed is dominated by network bandwidth. Benchmarks are very much application dependent since parallel efficiency is greatly affected by the lattice size, by the amount of computation performed per site, processor speed and type of interconnection. In many typical applications, like the one described in the preceding example, the drop in efficiency is less than 10% up to 8 nodes (processes) and less than 20% up to 32 (our tests are usually performed on a cluster of Pentium 4 PCs (2.2GHz) running Linux and connected by Myrinet).

# 5 MDP and PSIM

For portability reasons `MDP` is based on MPI. Nevertheless it is desirable to be able to run, test and debug `MDP` programs on a single node (with single or multi processor architecture) without having to install MPI. The latest version of `MDP` includes a Parallel SIMulator (PSIM). Despite the name this is not quite a simulator but an emulator, i.e. a message passing library that uses local (unix/posix) socket pairs. PSIM is Objected Oriented and is not based on MPI.

When compiling with PSIM, the parallel processes are created at start-up by forking. The number of parallel processes is specified at runtime by passing the following command line argument to any `MDP` executable program

```
-PSIM_NPROCS=4
```

(this makes 4 parallel processes).

PSIM also creates a communication log that can be used for debugging. `MDP` with PSIM has been tested on Linux, Mac and Windows (with cygwin).

For single processor node, using PSIM does not introduce any speed-up but, for a small number of processes (2-16) it does not slow down the code either. For multi-process shared memory architecture, parallelization of PSIM should produce a speed-up comparable with MPI. We have been yet performed such tests.

Moreover, PSIM should perform well on openMosix clusters as soon as openMosix starts supporting migratable sockets since all communication between the parallel processes will be done by the operating system. Unfortunately openMosix does not support migratable sockets yet.

# 6 Example: Ising Model

As one more example of usage of `MDP` we report here a simple program for the Ising model.