Parallel GPDT

A Parallel Gradient Projection-based Decomposition Technique for Support Vector Machines*

THOMAS SERAFINI[1] GAETANO ZANGHIRATI[2] LUCA ZANNI[1]

[1] *Department of Mathematics, University of Modena and Reggio Emilia, Italy*
E-mail: serafini.thomas@unimo.it, zanni.luca@unimo.it

[2] *Department of Mathematics, University of Ferrara, Italy*
E-mail: g.zanghirati@unife.it

Version 1.0 – Revision: July 2006

Overview

GPDT is a C++ software designed to train Support Vector Machines (SVMs) for binary classification problems in both scalar and distributed memory parallel environments. It solves the SVM convex quadratic programming (QP) problem by a decomposition technique and uses gradient projection methods for the inner QP subproblems.

Some important features are:

- designed to decompose into medium-to-large ($O(10^2) \div O(10^3)$) QP subproblems
- two gradient projection methods for the inner QP subproblems are available
- caching of kernel evaluations
- supports standard kernel functions and lets you define your own
- folding in the linear case
- sparse vector representation
- good efficiency with both small (O(10)) and large ($O(10^3)$) subproblems (medium-large subproblems are preferable)
- full SVM^{light}-compliant I/O

Moreover, the parallel version supplies:

- parallel implementation of the two gradient projection inner QP solvers
- parallel updating of the objective gradient
- distributed caching of kernel evaluations
- standard MPI communication routines

Algorithm

The decomposition technique implemented by GPDT follows the Joachims' SVM^{light} algorithm [1]. The main difference consists in the solver used for the inner QP subproblems. GPDT solves the subproblems by special gradient projection methods [3, 5, 6], which are very effective also on large subproblems ($O(10^3)$). This ability gives the chance to solve the whole problem in few decomposition iterations. To make effective such an approach, an appropriate working set selection is introduced [4]. Furthermore, to reduce kernel evaluations, special strategies are used for both gradient updating and kernel caching.

The parallel implementation is developed by distributing the main tasks of each decomposition iteration [2, 7]:

- building the subproblem Hessian;

*This software was developed with the support of the Italian Education, University and Research Ministry (grants FIRB2001/RBAU01JYPN and FIRB2001/RBAU01877P).

- solving the subproblem;
- updating the objective gradient.

For the first item, computation and storage of the needed entries are distributed block columnwise among the available processors. On such a distribution is also based the parallel inner QP solver, where the main task per iteration is given by a matrix-vector product involving the subproblem Hessian. In order to update the objective gradient some columns of the objective Hessian are required: these computations (requiring potentially expensive kernel evaluations) are distributed among the processors. Moreover, a given part of each processor memory is reserved as a caching area, where some Hessian columns are stored in order to avoid their recomputation in subsequent iterations.

Source Code

The program is distributed under the General Public License and available at

`http://dm.unife.it/gpdt`

also reachable starting from

`http://dm.unife.it/pn2o/software`

or

`http://slipguru.disi.unige.it/ASTA`

The sources for both the serial and the parallel versions are available to download. Binaries have presently been tested on the following systems:

- PC with all major Linux distributions (kernel 2.4 or higher, `gcc` 2.95 or higher): serial.
- PC with Windows ME, 2000, XP: serial.
- HP or DEC UNIX workstations: serial.
- Intel-based IBM Linux cluster (Xeon): serial and parallel.
- Power4- and Power5-based IBM AIX cluster: serial and parallel.

Building the program

To compile the sources you need a C++ compiler for your platform and a UNIX-like `make` command. Most Windows-targeted C/C++ programming IDE provide the GNU-like `gmake` command that can be called from the DOS prompt. Make the source directory your current directory, then edit the `Makefile` file and set the environment variable `$CPP` to the name of your C++ compiler command: for instance, type

```
$CPP = gcc
```

if you will use the standard GNU compiler. Then set the optional compiler command line arguments in the `$CPPFLAGS` environment variable. This is the right place where to put platform-specific code optimization options such as `-O3` or `-Ofast`. The variable `$SLIB` is mainly intended for expert users and shouldn't typically need any change from the provided defaults. Type in this variable the list of additional libraries that you want to be linked with the serial executable, in the standard way `-l`*libname*. Notice that *you always need* to link the standard math library by `-lm`.

All binaries will be created by default in the current directory, but you have a chance to change this behavior by setting the `$BINDIR` environment variable to provide a different directory (under UNIX-like systems you must previously check to have writing rights to that directory).

The compilation process should be quite fast and normally you shouldn't get errors. Eventually warning messages could be issued by the compiler if strict compliant controls are activated.

At the system prompt, simply type

```
make
```

or

```
make gpdt
```

to get the serial executable `gpdt` in the current directory. Analogously, the command

```
make pgpdt
```

will build the parallel version from the corresponding sources.

How to use

GPDT can be called by the following command line:

```
./gpdt [options] example_file model_file
```

The available options are:

option	argument	meaning
`-?`		this help
`-h`		display help message
`-v`	[0..2]	verbosity level (default 1)
`-t`	[0..2]	type of kernel function (default 2): 0: linear ($x^T y$) 1: polynomial $(s(x^T y) + r)^d$ 2: radial basis function (rbf): $\exp(-g\|\|x - y\|\|^2)$
`-s`	*float*	parameter s in polynomial kernel (default 1.0)
`-r`	*float*	parameter r in polynomial kernel (default 1.0)
`-d`	*int*	parameter d in polynomial kernel (default 3)
`-g`	*float*	parameter g in rbf kernel (default 1.0)
`-c`	*float*	parameter C for SVM classification: trade-off between training error and margin (default 10.0)
`-q`	*int*	size of the QP-subproblems: $q \geq 2$ (default 400)
`-n`	*int*	maximum number of new indices entering the working set in each iteration: $2 \leq n \leq q$, n even (default $q/3$)
`-e`	*float*	tolerance for termination criterion (default 0.001)
`-a`	$[0, 1]$	gradient projection-type inner QP solver: 0: Generalized Variable Projection method [3] 1: Dai-Fletcher Projected Gradient method (default) [5]
`-f`	$[0, 1]$	projector type [7]: 0: bisection-like algorithm (default if `q` <= 20) 1: secant-based method (default if `q` > 20)
`-m`	*int*	cache size in MB (default 40)
`-u`	*float*	parameter for proximal point modification [7] (default 0): `u` > 0: modification at each iteration `u` < 0: modification, with parameter `abs(u)`, only as emergency step

Remember that different actions will be automatically taken if you provide invalid parameter values. Anyway, at the end of the run you will be informed on the values actually used by the code.

Testing the program

Here we provide some execution examples. Three test sets are available:

- a 10000 samples subset of the MNIST handwritten digits database (`http://yann.lecun.com/exdb/mnist`): it is constituted by 5000 samples of the digit "8" and 5000 samples of the other digits (file `mnist8n8_10k`);
- the 11220 samples set "adu6" of the UCI Adult database at `http://www.research.microsoft.com/~jplatt` (file `uciadu6`);
- the 17188 samples set "web-6a" of the Web database at `http://www.research.microsoft.com/~jplatt` (file `web6a`).

In what follows we show a sample command line typed at the system prompt and the training results.

Scalar tests

The following tests are carried on an HP zx6000 workstation (CPU Intel Itanium 2 at 1.3GHz, 2GB RAM).

For instance, the command line

```
./gpdt -t 2 -g 1.54e-7 -c 10 -q 1000 -n 400 -m 300 mnist8n8_10k mnist8.model
```

will train a Gaussian SVM (in this case the option `-t 2` can be omitted, since it is the default) with $\sigma = 1800 \Rightarrow g = 1.54 \cdot 10^{-7}$ and $C = 10$ on the 10000 samples MNIST data set, using a working set sized 1000, allowing at most 400 new variables entering the working set at each decomposition iteration and requiring a 300 MB caching area. The classifier will be stored in the file `mnist8.model`.

Table 1 shows the training reports on the three data sets with different decomposition subproblem sizes.

Data set	-q	-n	obj	b	SV	BSV	iter.	ker. ($\times 10^6$)	sec.
MNIST (rbf kernel with $\sigma = 1800$, $C = 10$)	2	2	-2083.574	6.4389	1589	39	10263	17.032	41.7
	40	20	-2083.575	6.4399	1591	39	760	18.289	21.0
	400*	132*	-2083.576	6.4400	1590	39	55	19.105	20.6
	1000	400	-2083.576	6.4401	1591	39	14	21.224	24.4
UCI Adult (rbf kernel with $\sigma = \sqrt{10}$, $C = 1$)	2	2	-3750.461	-0.0897	4212	3741	6752	50.946	28.2
	40	20	-3750.461	-0.0901	4219	3723	559	51.826	11.9
	400*	200	-3750.462	-0.0900	4226	3715	35	54.664	11.1
	1000	500	-3750.461	-0.0889	4226	3715	15	58.476	13.0
Web (linear kernel with $C = 1$)	8	4	-536.967	1.0269	820	475	16437	0.493	113.0
	40	20	-536.968	1.0268	814	474	3024	1.905	21.2
	200	100	-536.969	1.0268	809	478	114	1.422	4.3
	800	400	-536.967	1.0277	792	479	13	1.572	11.7

* Predefined default values.

Table 1: numerical results for some serial runs on the three provided test sets.

Parallel tests

To test the parallel version, once the appropriate binary is created by compilation, you can proceed as in two following examples. Note that the scripting syntax and the job submission system can be quite different on your specific platform, so in case of troubles please ask the technicians or browse the available system documentation.

Suppose you are a user of a Power4-based IBM cluster and that the executable `pgpdt` is available. If your machine supports POE (Parallel Operating Environment), the following line will run the program in interactive mode on 8 processors of the same node, with the same requirements as for the corresponding scalar test:

```
poe ./pgpdt -procs 8 -nodes 1 -task_per_node 8 \
            -t 2 -g 1.54e-7 -c 10 -q 400 -n 132 -m 300 \
            mnist8n8_10k mnist8.model > mnist8n8_10k.out
```

where the standard output is redirect to the file `mnist8n8_10k.out` in the current directory.

Since it is often strongly discouraged to run demanding parallel jobs in interactive mode, more reliable tests can be carried on in batch mode. The following is a sample command script to submit the same test in batch mode:

```
#!/bin/sh
# @ wall_clock_limit = 0:10:00
# @ network.MPI = csss,shared,US
# @ node = 1
# @ tasks_per_node = 8
# @ resources = ConsumableCpus(1) ConsumableMemory(1500 mb)
# @ shell = /bin/csh
# @ job_name = SVMPAR
# @ job_type = parallel
# @ class = debug
# @ output = svm.out
# @ error = svm.err
# @ notification = never
# @ queue

cd $HOME/pgpdt

./pgpdt -t 2 -g 1.54e-7 -c 10 -q 400 -n 132 -m 300 \
        mnist8n8_10k mnist8.model > mnist8n8_10k.out
```

It should be saved in a file (say `mnist8n8_10k.cmd`) and submitted to the batch queue. Please, refer to your specific site documentation for further details on policies, configurations, requirements, script syntax and machine limits.

Conversely, suppose you are a user of a Xeon-based Linux cluster, that the executable `pgpdt` is available and that the MPICH library is installed on the system. If your machine supports OpenPBS (Open Parallel Batch System), then normally the only possibility to run MPI-based parallel programs is through the command `mpiexec`, both in a batch interactive and in a batch background sessions.

If you want to run `pgpdt` interactively on 8 processors then you could use the following sequence of commands:

```
qsub -l nodes=4:ppn=2,walltime=0:10:00 -I
```

Data set	Procs	obj	b	SV	BSV	iter.	ker. ($\times 10^6$)	sec.
MNIST	1	-2083.575	6.4397	1590	39	57	19.095	28.7
($\sigma = 1800$, $C = 10$	2	-2083.576	6.4403	1589	39	53	19.469	15.1
$q = 400$, $n = 132$,	4	-2083.575	6.4400	1590	39	52	19.878	8.6
cache = 300MB)	8	-2083.576	6.4402	1590	39	54	20.679	5.6

Table 2: numerical results of some parallel runs on a Xeon-based IBM Linux cluster.

At this point (if there are free resources) you enter the batch interactive session and you can run your test with:

```
mpiexec ./pgpdt -t 2 -g 1.54e-7 -c 10 -q 400 -n 132 -m 300 \
                mnist8n8_10k mnist8.model > mnist8n8_10k.out
```

The same can be done in a non-interactive batch session: you should save in a file, say `mnist8n8_10k.sub`, a job script similar to

```
#!/bin/sh
#PBS -l nodes=4:ppn=2,walltime=00:10:00

cd $HOME/pgpdt

mpiexec -n 8 -no-shmem \
        ./pgpdt -t 2 -g 1.54e-7 -c 10 -q 400 -n 132 -m 300 \
                mnist8n8_10k mnist8.model > mnist8n8_10k.out
```

and then submit it to the queuing system with the command

```
qsub mnist8n8_10k.sub
```

Please, refer to the specific documentation of your parallel Linux system for more detailed information.

Table 2 summarizes the expected results on the MNIST data set by changing the number of processors. We underline that much better scaling performance can be obtained by training larger data set, as it is shown for instance in [2].

Also, notice that you could observe a variability in the execution time on repeated runs of the same command line, due to both the user policy and the machine workload.

References

[1] T. Joachims (1998), Making Large-Scale SVM Learning Practical, *Advances in Kernel Methods – Support Vector Learning*, B. Schölkopf, C.J.C. Burges and A. Smola, eds., MIT Press, Cambridge, Massachusetts.

[2] G. Zanghirati, L. Zanni (2003), A Parallel Solver for Large Quadratic Programs in Training Support Vector Machines, *Parallel Computing* **29**, 535–551.

[3] T. Serafini, G. Zanghirati, L. Zanni (2005), Gradient Projection Methods for Large Quadratic Programs and Applications in Training Support Vector Machines, *Optimization Method and Software* **20**, 353–378.

[4] T. Serafini, L. Zanni (2005), On the Working Set Selection in Gradient Projection-based Decomposition Techniques for Support Vector Machines, *Optimization Method and Software* **20**, 583–596.

[5] Y.H. Dai, R. Fletcher (2005), New Algorithms for Singly Linearly Constrained Quadratic Programming Problems Subject to Lower and Upper Bounds, *Matematical Programming* **106**(3), 403–421. Also published as Research Report NA/216, Dept. of Mathematics, University of Dundee, UK.

[6] L. Zanni (2006), An Improved Gradient Projection-based Decomposition Techniques for Support Vector Machines, *Computational Management Science* **3**(2), 131–145.

[7] L. Zanni, T. Serafini, G. Zanghirati (2006), Parallel Software for Training Large Scale Support Vector Machines on Multiprocessor Systems, *JMLR* **7**(Jul), 1467–1492 (Special Topic on Machine Learning and Optimization). Also published as TR 356, Feb. 2006, Department of Mathematics, University of Ferrara, Italy.