Nuclear Engineering Laboratory
M O N T E C U C C O L I N O
technical report
Nuclear Reactor Physics
Department of Energy and Nuclear Engineering
and of Environmental Control (DIENCA)
University of Bologna

Doc. ID: LIN-R04.2007
Subject: es-cPIF profiling
Date: May 25, 2007

Author: Giacomo Grasso
Phone: +39-051-6441708
e-mail: giacomo.grasso@mail.ing.unibo.it

Profiling results for *es-c*PIF optimization

The ***es-c*PIF** (***e***lectro***s***tatic-***c***ollisional **P**article **I**n cell for plasma **F**ocus devices) code has been developed with High Performance Computing (HPC) wills. To achieve this goal, the code writing has been deeply analyzed on two levels: a direct one of program main blocks, functions and operations architecture optimization, and an indirect second one of cache memory access optimization by means of advanced particles management techniques.
The developed ***es-c*PIF** code efficiency has then been tested by low level profiling by means of the IBM's Hardware Performance Monitor (HPM) tool. The obtained results are here presented and compared with the ones of a BLAS benchmarking code suitably written.

The Author
Giacomo Grasso

The Supervisor
prof. Marco Sumini

Revisions List:

Date	Author

Profiling results for ***es-c*PIF** optimization

Giacomo Grasso

May 25, 2007

Abstract

The ***es-c*PIF** (***e***lectro***s***tatic-***c***ollisional **P**article **I**n cell for plasma **F**ocus devices) code has been developed with High Performance Computing (HPC) wills. To achieve this goal, the code writing has been deeply analyzed on two levels: a direct one of program main blocks, functions and operations architecture optimization, and an indirect second one of cache memory access optimization by means of advanced particles management techniques.
The developed ***es-c*PIF** code efficiency has then been tested by low level profiling by means of the IBM's Hardware Performance Monitor (HPM) tool. The obtained results are here presented and compared with the ones of a BLAS benchmarking code suitably written.

Contents

1 Introduction

The development of a new code for numerical analysis aiming at HPC status requires a deep investigation phase for tuning its execution at the highest speed. Even in order to parallelize the code, it's fundamental that the kernel executes at best as it is possible: it's vain to multiply the computing capabilities of a code if every clone wastes time in useless calculations.

For a code such as ***es-c*PIF** [1, 2], managing enormous volumes of data by repetitive operations, complicatedly articulated in a many levels "if" tree, the reduction of the branches number, the optimization of the numerical calculations and the efficient management of hardware memories are concrete priorities in the optimization of the whole program.

On the other side, the modelled physics imposes several constrictions on the code overall structure, on the order of the involved operations and on the particles management (to whom memory is strictly connected).

Extreme attention has therefore been paid to an efficient code writing. To verify the effectiveness of the work, the code has then been analyzed by means of the IBM's HPM [3] profiling tool, included within the AIX 5.3 installation onto the **sp5** cluster at CINECA.

2 Inclusion of HPM calls into the source code

To compile a generic program with HPM library support, several changes are to be brought to the source code and the simulation environment, summarizable in a three-steps procedure here described (all the sample commands reference to a fortran code as an example: only minor syntactic changes are needed indeed for C++ codes).

2.1 Library initialization and sections definition

The first fundamental change in the main program file of the code is to add the header inclusion instruction

```
#include "f_hmp.h"
```

to define the external monitoring functions used in the code.

It is then possible to initialize the hardware counters by calling the proper subroutine before any specific section to be monitored:

```
CALL f_hpminit( n, "string" )
```

where

- `n` is an integer index indicating the number of initialization of the library (usually 0);
- `string` is a textual label to identify within the output file the results produced by the specific initialization of the library.

The same index is needed at the end of the monitored section to finalize the hardware counters by calling the analogue subroutine:

```
CALL f_hpmterminate( n )
```

The last modify in the source code is the inclusion of the instructions for beginning and stopping the hardware counters; the code is therefore sectioned by means of couples of instructions like

```
CALL f_hpmstart( m, "section label" )
CALL f_hpmstop( m )
```

As for the initializing and finalizing functions, it is possible to specify a progressive, unique integer index for each section, as well as a textual label to make more readable the output.

Like "do loops", monitored sections may be nested each other, but never "cross" ended.

2.2 Choice of the hardware counters to be used

The HPM library has the capability to monitor a wide set of hardware counters and performance indexes, from main memory and different levels caches management (data hit, miss, bandwith, etc.) to effective operations per CPU cycle. The user can therefore choose between several sets of alike counters groups to focus on several related aspects of performance monitoring.

The library retrieves the informations about the counters sets to activate by means of the environment variable `HPM_EVENT_SET`: every set is identified by an integer, and the `HPM_EVENT_SET` variable may be initialized with a list of integers referring to the specific sets to be activated (the default value of the `HPM_EVENT_SET` variable is "5").

The user can therefore specify the desired counters sets by creating in the program main directory the text file `HPM_flags.env` which redefines the environment variable `HPM_EVENT_SET` used by the library. The presence of the `HPM_flags.env` file overrides the existing value of the `HPM_EVENT_SET` variable.

The chosen counters sets (a list of which can be retrieved from the HPM documentation [3]) must be in a comma separated, space free list (i.e. "6,4,7,8,5").

2.3 Program compilation and linking

After the source code modifies of section 2.1 the fortran code is now ready to be compiled. On IBM machines, it must be done by using the owned xl∗ compilers. The compiling instruction to use is the following (e.g. xlf_r fortran compiler):

```
xlf_r -c -qfree=f90 -qsuffix=cpp=f90 -I /usr/include main_src_file.f90
```

(for a f90 source code format, and a f90 source files extension). Any module or subroutine file must be compiled in the standard way.

The objects produced can be finally linked by specifying the libraries to be used, with:

```
xlf_r lpmapi -lm -lessl -o progname main_src_file.o
```

3 Results

The results here presented refer mainly to the cache access efficiency. The effective number of operations per CPU cycle cannot be considered for benchmarking with the BLAS case due to the high number of "if" branches needed by the physical problem modeling.

The HPM output for the ***es-c*PIF** code follows: five sections have been identified and separately measured:

1. Main section: is the whole program (the only file input and output phases have been excluded) one;

```
Instrumented section: 1 - Label: Main Program - process: 0
file: Main.f90, lines: 75 <--> 330
 Count: 1
```

```
 Wall Clock Time: 114613.393926 seconds
 Total time in user mode: 113262.226490389 seconds
 Exclusive duration: 79.306 seconds
```

2. Time loop: this section measures the whole computing time for the program main loop;

```
Instrumented section: 2 - Label: Time loop - process: 0
file: Main.f90, lines: 191 <--> 295
 Count: 1
 Wall Clock Time: 114534.087945 seconds
 Total time in user mode: 113186.301160638 seconds
 Exclusive duration: 7774.67 seconds
```

3. Reshape: this section is intended to measure the computing time required for the counting sort and the merging operation, for a efficiency balance with respect to the saved time in the code kernel;

```
Instrumented section: 3 - Label: Reshape - process: 0
file: distributer.f90, lines: 351 <--> 669
 Count: 2325
 Wall Clock Time: 86836.8083839999 seconds
 Total time in user mode: 86035.6697793655 seconds
 Average duration: 37.3492
 Standard deviation: 5.326e-315
```

4. Particles push: this section is the generic code kernel, including the nodes to particles field computing, the particles push and the MCC module;

```
Instrumented section: 4 - Label: Particles push - process: 0
file: mover.f90, lines: 69 <--> 187
 Count: 2325
 Wall Clock Time: 19922.605505 seconds
 Total time in user mode: 19757.3237857531 seconds
 Average duration: 8.56886
 Standard deviation: 5.31323e-315
 Exclusive duration: 7184.01 seconds
```

5. Electrons push: is the analogue to the previous section but reduced to the only electrons case.

```
Instrumented section: 5 - Label: Electrons push - process: 0
file: mover.f90, lines: 70 <--> 125
 Count: 2325
 Wall Clock Time: 12738.591463 seconds
 Total time in user mode: 12615.3464648492 seconds
 Average duration: 5.47896
 Standard deviation: 5.31023e-315
```

The global performances of the code can be obtained by the results of section 1:

```
Utilization rate                                      :         98.799 %
MIPS                                                  :       1618.246 MIPS
Instructions per cycle                                :          0.860
Total L2 data cache accesses                          :     290421.689 M
```

```
% accesses from L2 per cycle                                          :          0.135 %
L2 traffic                                                            :   35451866.323 MBytes
L2 bandwidth per processor                                            :        309.317 MBytes/s
Total load and store operations                                       :  111620364.027 M
Instructions per load/store                                           :          1.662
Number of loads per load miss                                         :       1193.601
Number of loads per DTLB miss                                         :      32956.237
Number of stores per store miss                                       :        117.833
Number of loads per TLB miss                                          :      34575.984
Number of load/store per TLB miss                                     :      44940.864
Memory load traffic                                                   :   52882646.812 MBytes
Memory bandwidth per processor                                        :        461.400 MBytes/s
Number of loads from local memory per loads from remote memory        :          2.649
% loads from memory per cycle                                         :          0.002 %
```

By comparison with the BLAS benchmark case ones

```
Utilization rate                                                      :         97.306 %
MIPS                                                                  :       4497.830 MIPS
Instructions per cycle                                                :          2.427
Total L2 data cache accesses                                          :       1882.425 M
% accesses from L2 per cycle                                          :          2.796 %
L2 traffic                                                            :     229788.233 MBytes
L2 bandwidth per processor                                            :       6325.048 MBytes/s
Total load and store operations                                       :      50216.512 M
Instructions per load/store                                           :          3.254
Number of loads per load miss                                         :        108.511
Number of loads per DTLB miss                                         :       1117.182
Number of stores per store miss                                       :          0.419
Number of loads per TLB miss                                          :       1035.299
Number of load/store per TLB miss                                     :       1047.745
Memory load traffic                                                   :     338908.406 MBytes
Memory bandwidth per processor                                        :       9328.642 MBytes/s
Number of loads from local memory per loads from remote memory        :         26.106
% loads from memory per cycle                                         :          0.032 %
```

it can be seen how the memory management is improved, increasing the number of memory load/store operations per miss fault and consequently reducing the accesses percentage from L2 cache per cycle and the loads percentage from memory per cycle; on the other side, the resulting number of instructions per cycle is reduced by consequence of the "if" branches.

4 Concluding remarks

The optimizations introduced in the latest versions of the ***es-c*PIF** code [2] by means of rearrangement of simulation particles in memory [4], maximization of consecutive operations performed on every particle and generic numerical techniques tuning [1] let the code gain performances in reducing useless operations and memory traffic. This results in a reduced computational time by a factor 4.5.

Moreover, the HPM profiling results show an averaged 50% performance increase with respect to the well performing, benchmark BLAS case in memory management. The same is not valid with respect to the number of effective operations per CPU cycle, due to the

physical phenomenon modeled: the statistical nature of the system requires a high number of "if" branches for the control of the particles behavior, resulting in a sensible time waste.

References

[1] G. Grasso. Applicazione di metodi article In Cell alla modellizzazione della dinamica del plasma in dispositivi Plasma Focus. Master's thesis, University of Bologna, 2006.

[2] M. Frignani. *Simulation of Gas Breakdown and Plasma Dynamics in Plasma Focus Devices*. PhD thesis, University of Bologna, 2007.

[3] IBM. *Performance Tools Guide and Reference*, 3^{rd} edition, July 2006.

[4] J. K. Bowers. Accelerating a Particle-In-Cell simulation using a hybrid counting sort. *J. Comp. Phys.*, 173(2):393–411, 2001.