Weather Research and Forecast Program Analysis and its General Consequences

Luděk Kučera

Czech Technical University & Charles University
Prague, Czech Republic
`ludek@kam.mff.cuni.cz`

Abstract. The arithmetic intensity of the most important parts of the Weather Research and Forecast program was measured and it appears to be below 0.5 Flops/byte, not more than about twice the arithmetic intensity of the HPCG benchmark. This suggest that the most of the recent supercomputers would not be use more than about 3-4 % their peak computing power when running WRF (and it can be expected that a similar bounds will apply to other programs in meteorology and climatology as well).

Keywords: sparse matrix, HPCG, computer architecture

1 Introduction

WRF (Weather Research and Forecast) program is an open source program for atmospheric physics. Its development started at and it represents now one of the most important tools in meteorology, being actively used, e.g., by ... . The present paper brings some preliminary results on computational behavior of WRF.

However, the (main) aim of the paper is not presenting data about computational aspects of WRF. Our principal goal is to find out whether the arithmetic intensity (the flop/byte rate) of WRF is likely to be below 0.5-1.0.

The main characteristic of supercomputers, used for their ranking in Top500, is the LINPACK behavior. This is based on the assumption that it gives the assessment of *practical* use of a system, because LINPACK package belongs to standard libraries of Computational Linear Algebra, one of the main sources of the HPC software.

LINPACK is well correlated with the peak computing power, being usually about 50 - 80 % of the peak. Thus, the peak ranking would be very similar to the LINPACK one, and hence there is no real need to replace LINPACK by the peak ranking. (Nevertheless, mainly in connection with reaching the exascale goal, there are some first attempts to adopt the peak as the main indicator of the computing power.)

However, there is growing feeling among certain HPC scientists that the LINPACK benchmark, even though a part of the linear algebra libraries, does not reflect well the true computing power of supercomputers, because the nature of most HPC problems that are routinely solved in supercomputing centers is quite different from the nature of LINPACK, mainly because LINPACK is a dense matrix routine, while typical problems solved in practice use very sparse matrices.

This is why ... and J. Dongarra developped another benchmark, High Performance Conjugate Gradient (HPCG) that is based on the Conjugate Gradient Method (CGM), which is one of the principal iterative methods of solving sparse systems of linear equations. (Note that LINPACK, being essentially a linear equation solver based on Gaussian elimination, can not be applied to large sparse matrices, because even a sparse matrix of a linear system becomes dense during Gaussian elimination, and it would not fit into the computer's memory).

The HPCG benchmark was introduces in 2014 and the result (see [] for all available lists) are extremely disappointing: in general, recent large system use only about 1-2 % of their peak computing power (!!!), see Fig 1. This suggests that the recent HPC systems might not be now well suited for the problems that are solved with their use.

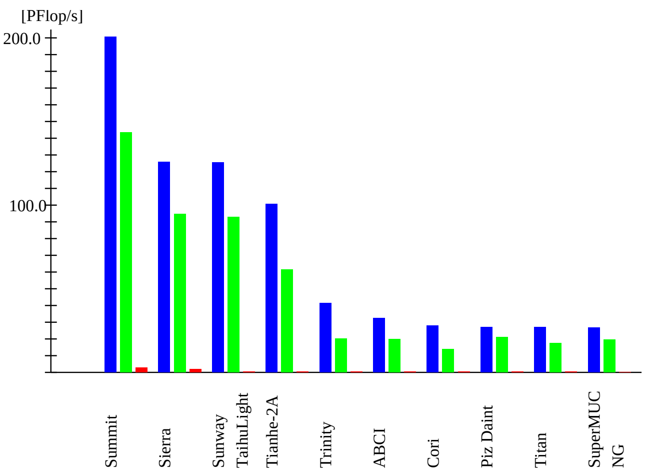

Fig. 1: HPCG

The HPCG benchmark was developped to model applications that cover *most of the time* spent by HPC simulations[1]. If HPCG really has reflected the behavior of the most simulation programs were true, then it would have been appropriate to consider developping a new line of the HPC hardware that would process sparse matrix simulations more efficiently.

[1] deap learning and tensor processor units

Nevertheless, even though HPCG results were included as one column into the Top500 list table, they receive little attention. For example, practically nobody has welcome Summit as the *first* supercomputer that crossed 1 PFlop/s HPCG barrier.

ut seems that HPCG is regarded as yet another artificial benchmark

The *Weather Research and Forecast* program (WRF) is, as the name suggests, designed for simulation of meteorologic situations. While the kernel of the HPCG is a very small and homogenous piece of code, WRF is a very complex program to somulate a very complex physical situation. Unlike in HPCG, and also unlike in the most CFD codes, the medium modelled by WRF, the air, is not a simple mixture of O2, NO2, and CO2; the main complexity comes from the presence of water that occurs in the form of moisture, fog, rain drops, show, groats, etc. Moreover, long wave and short wave radiation must be considered to determine correctly temperature. Another very complex factor is the Earth surface (land, see). Thus, it can be expected that the amount of computation corresponding to one node of the grid is substantially higher than in the case of HPCG, which has potential to increase the arithmetic intensity of the computation.

un the WRF, the physics of atmosphere is divided to several drivers that are called from the module *module_integrate*, among them the microphysics driver, the radiation driver, the surface driver, the planetary boundary layer (PBL) driver, the cumulus driver, and some others.

The following table shows

Driver	Time [%]
microphysics driver	19.9
radiation driver	8.0
surface driver	5.2
PBL driver	3.8
cumulus driver	3.9

The total of the above drivers is slightly more that 40 % of the time.

un the table, the first column gives the total number of time spent executing the subroutine and subroutines that it calls directly or indirectly; note that 1 % ot the total computing time was slightly more than 10 second in our experiment. The second column shows how many times a subroutine was called.

Timing of calls was done using calls to SYSTEM_CLOCK. It has been found that one clock call (either at the beginning or at the end of a subroutine execution) needed about 1.2 microseconds. The table has been used only to select the most time consuming parts of the WRF, and the choice would remain unchanged is the clock call times were subtracted (note that the subroutine of the table with the shortest execution time is $sflx$ - about 9 microseconds).

```
TIME [\%]   FREQ.
100.0       1 wrf
 99.1       1 |  module_integrate/integrate()
 99.9     240 |  |  module_integrate|solve_interface()
 19.9     240 |  |  |  module_microphysics_driver/microphysics_driver/()
 19.9     240 |  |  |  |  module_mp_wsm3/wsm3()
 24.9     240 |  |  |  module_first_rk_step_part1/first_rk_step_part1()
  3.6     240 |  |  |  |  module_big_step_utilities_em/phy_prep()
  8.0     240 |  |  |  |  module_radiation_driver/radiation_driver()
  7.5      24 |  |  |  |  |  module_ra_rrtm/rrtmlwrad()
  7.4  503616 |  |  |  |  |  |  module_ra_rrtm/rrtm()
  5.2     240 |  |  |  |  module_surface_driver/surface_driver()
  3.8     240 |  |  |  |  |  module_sf_noahdrv/lsm()
  3.4 4100640 |  |  |  |  |  |  module_sf_noahlsm/sflx()
  3.8     240 |  |  |  |  module_pbl_driver/pbl_driver()
  3.1     240 |  |  |  |  |  module_bl_ysu/ysu()
  2.9   29280 |  |  |  |  |  |  module_bl_ysu/ysu2d()
  3.9     240 |  |  |  |  module_cumulus_driver/cumulus_driver()
  3.9     121 |  |  |  |  |  module_cu_kfeta/kf_eta_cps()
  3.2 2458178 |  |  |  |  |  |  module_cu_kfeta/kf_eta_para()
  2.8     240 |  |  |  module_first_rk_step_part2/first_rk_step_part2()
  5.9     720 |  |  |  module_em/rk_tendency()
  3.3    3120 |  |  |  module_small_step_em/advance_uv/()
 15.1    2160 |  |  |  module_em/rk_scalar_tend()
 11.7     720 |  |  |  |  module_advect_em.F/advect_scalar_pd()
  2.7     240 |  |  |  module_big_step_utilities_em/moist_physics_prep_em/()
 10.1     960 |  |  |  module_big_step_utilities_em/calc_p_rho_phi/()
```

Table 1: Time and frequency of the main subroutine calls

The table is a selection from the report of the execution times of 259 subroutines (among them 169 subroutines with the time share smaller tham 0.1 % and 48 others with the share below 1 %).

We have not studied the arithmetic intensity of the whole computation, but we have selected 3 main parts of the code with the most important time share: the microphysics driver that has exceptional position among parts of WRF, the radiation driver as the most time consuming among the other physics driver, and then the subroutine *rk_scalar_tend* as the principal among other subroutines. All of them are still composed from a larger number of components, among which one is much more important than the others, and hence finally the following subroutines have been selected for the detailed investigation:

- *WSM3* from the microphysics driver (phys/module_mp_wsm3.F),
- *RRTMLWRAD* from the radiation driver (phys/module_ra_rrtm.F), and
- *advect_scalar_pd*, the principal subroutine called by *rk_scalar_tend* (dyn_em/module_advect_em.F)

For each of the three tested subroutines (together with subroutines that are directly or indirectly called by them) we measured the number of basic arithmetic operations (addition, subtraction, multiplication, division), certain functions as *min*, *max*, *abs*, and more complex functions as *exp*, *log*, *sqrt*. Operations used to evaluate conditions of *IF* statements, control and bounds of *DO* statements were not considered, because it has been found that thay represent only a minor portion of all operations.

Moreover, any variable that was used to evaluate right hand side of an assignment statement represented one elementary data transfer from the cache or the memory into the arithmetic unit (load) and any variable representing the left hand side of an assignment represented one lementary data transfer from the arithmetic unit into the cache or the memory (store).

As already pointed above, our definition arithmetic intensity involves only transfers between the arithmetic unit and the memory, but not between the unit and the cache.

The numbers of arithmetical and logical operations performed by the program is uniquely determined by the program and input data, and it does not depend ona particular computing hardware. They are given (in millions) in the following three tables (for wsm3, RRTMLWRAD and advect_scalar_pd).

Table 1: WSM3: numbers of operations [in millions]

add/sub	mul	div	min/max	exp	log	sqrt	**loads**	**stores**
57.9	81.4	28.6	39.8	14.5	9.1	2.4	**152.2**	**93.3**

Table 2: RRTMLWRAD: numbers of operations [in millions]

add/sub	mul	div	min/max	exp	log	**loads**	**stores**
2944.7	1581.9	158.3	3.5	2.1	0.9	**4371.1**	**1456.8**

Table 3: advect_scalar_pd: numbers of operations [in millions]

add/sub	mul	div	min/max	sign	abs	**loads**	**stores**
51.6	39.7	8.4	7.1	1.7	3.6	**48.8**	**6.93**

The tables give also data transfers between arithmetic unit and the memory or caches. We do not consider scalar variables, because their number is relatively small and does not affect the numbers too much. We also do not consider moving parameters during function calls and data movements that are connected with arithmetic and logical operations that are not considered (loop control, if and while conditions. Note that in this way the number of executed calls is higher than the figures presented lebow, which means that the "efficiency" of computers running WRF is even lower than suggested in the paper.
ut is very important that our definition of arithmetic intensity considers processor - memory data transfers only, because the main source of low computing efficiency is caused by insufficient bandwidth of memory channels, while the bandwidth of the channels between an arithmetic unit(s) and caches is much higher.
As it is our goal to investigate potential architectures that would be more efficient when running sparse matrix simulations, we did not try to detect what data were stored in caches at any existing computer or a processor line. Instead, a very simple theoretical architecture and algorithm were used to evaluate the number of data transfers that must have crossed the processor/memory boundary. The memory model has two parts:

- a memory, formally a finite set of elements, called *variables*; the memory models a DRAM memory, including HBM's like MCDRAM of Intel Xeon or ... of Volta-100 (i.e., a memory that is connected to the processing unit via a non-silicon bus)
- a cache, formally a finite set of elements, called *cache locations*; in most cases, the cache informally corresponds to L1 and L2 caches (i.e., caches that are on the logic die of a processor chip or package)

un our formal model,
a variable v is an object that has an attribute $v.value$, the value of the variable, as stored in the memory, and
a cache location ℓ is an object that has four attributes ($\ell.value$, $\ell.mirrors$, $\ell.time$ having meaning only if $\ell.var \neq$ **null**):

- $\ell.var$ is either **null** (free location), or a variable (the one stored in the location);
- $\ell.value$; it need not generally be $\ell.var.value = \ell.value$, although this is the state that we would like to keep as often as possible;
- $\ell.mirrors$ is a boolean variable; it says whether the equality $\ell.var.value = \ell.value$ is guaranted (but the equality could hold even if $\ell.mirrors$ is **false**);
- $\ell.time$ denotes the time, when the value of $\ell.var$ has been changed for the last time.

We require that for each variable v there is at most one cache location ℓ such that $\ell.var = v$ (each variable is in the cache at most once). In the case when $\ell.var.value \neq \ell.value$, we assume that the valid value is that from the cache ($\ell.var.value$), while the value in the memory is out-dated, as indicated by $\ell.mirrors$ being **false**.
From the point of our memory model, a computation is a sequence $i_1, \ldots, i_k$, where each element of the sequence is either $L(v)$ (load the value of the variable v into the processing unit), or $S(v, \nu)$ (store the value ν to the memory or to the cache). We will also use two counters $nofloads$, $nofstores$, initially equal to 0, to count the number of loads/stores from/to the memory (not the cache).
We say that a cache location is *free* if $\ell.var$ is **null**. We say that the cache is *full* if there are no free cache locations. We will also use the procedure $make\_free\_location()$, called when the cache is full, and working as follows:

```
choose a cache location $\ell$;
if (!$\ell.mirrors$) {$\ell.var.value = \ell.value$; $nofstores++$; }
$\ell.var =$ null
```

i.e., some cache location is made free, and if it is not sure that the corresponding value in the memory is up-to-date, $\ell.value$ must be moved to the memore and the counter $nofstores$ incremented.
Now, a sequence of instructions mentioned above is processed in the following way:
The k-th instruction $L(v)$:

```
if(there is no cache location $\ell$ such that $\ell.var == v$) {
        if(the cache is full) $make\_free\_location()$;
        choose a free cache location $\ell$;
        $\ell.var = v$;
        $\ell.var.value = v.value$;
        $\ell.mirrors =$ true;
        $\ell.time = k$;
        nofloads ++;
}
The $k$-th instruction is $S(v, \nu)$
if(there is no cache location $\ell$ such that $\ell.var == v$) {
        if(the cache is full) $make\_free\_location()$;
        choose a free cache location $\ell$;
        $\ell.var = v$;
        $\ell.time = k$;
} else $\ell$ is the unique cache location such that $\ell.var = v$;
$\ell.var.value = v.value$;
$\ell.value = \nu$;
$\ell.mirrors =$ false;
```

Note that in the case of loading a variable that is in the cache, the cache value is simply loaded to the processing unit regardless of the update state of the memory copy and no other modification of the memory system is needed. In the case of store, the value is simply stored in the cache with the provision that it is not (any more) guaranteed that it mirrors the corresponding value in the memory. We need to store into the memory only when kicking from the cache a value that is not guaranteed to be a true copy of a memory value.

Table 4: WSM3: Effects of the cache

cache	0	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
loads	152.2	142.2	129.9	111.9	92.6	89.8	89.6	88.4	85.6	82.5	81.0	80.4	80.3	80.2	80.1	79.5	72.5	58.7	33.4
stores	93.3	90.9	86.1	83.7	83.7	83.2	82.9	82.7	82.7	81.1	74.8	67.9	65.8	65.0	64.6	64.3	63.7	59.9	42.8
all	245.5	233.1	216.0	195.7	176.3	173.0	172.5	171.1	168.2	163.6	155.8	148.3	146.1	145.2	144.7	143.8	136.3	118.6	76.2
load rate	1.000	0.934	0.854	0.735	0.608	0.590	0.588	0.581	0.562	0.542	0.532	0.529	0.527	0.527	0.526	0.522	0.477	0.386	0.220
store rate	1.000	0.974	0.923	0.898	0.897	0.892	0.889	0.886	0.886	0.870	0.802	0.728	0.706	0.697	0.693	0.690	0.683	0.642	0.459
all rate	1.000	0.949	0.880	0.797	0.718	0.705	0.703	0.697	0.685	0.667	0.635	0.604	0.595	0.591	0.589	0.586	0.555	0.483	0.310
flop/byte	**0.119**	**0.125**	**0.135**	**0.149**	**0.166**	**0.169**	**0.169**	**0.171**	**0.174**	**0.179**	**0.188**	**0.197**	**0.200**	**0.201**	**0.202**	**0.203**	**0.214**	**0.246**	**0.384**

Table 5: RRTMLWRAD: Effects of the cache

cache	0	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
loads	4371.1	3817.1	3263.2	2403.7	1977.0	1770.0	1662.8	1520.8	1367.4	1289.3	1256.4	1230.5	1107.6	1044.9	975.8	875.5	715.6	711.5	12.6
stores	1456.8	1456.8	1445.0	1453.1	1419.6	1412.9	1410.1	1407.8	1405.5	1405.0	1399.8	1234.1	927.9	839.4	797.4	772.1	772.7	759.3	5.7
all	5827.9	5273.9	4708.2	3856.8	3396.6	3182.9	3072.9	2928.6	2772.9	2694.3	2656.2	2464.6	2035.5	1884.3	1773.2	1647.6	1488.3	1470.8	18.3
load rate	1.000	0.873	0.747	0.550	0.452	0.405	0.380	0.348	0.313	0.295	0.287	0.282	0.253	0.239	0.223	0.200	0.164	0.163	0.003
store rate	1.000	1.000	0.992	0.997	0.974	0.970	0.968	0.966	0.965	0.964	0.961	0.847	0.637	0.576	0.547	0.530	0.530	0.521	0.004
all rate	1.000	0.905	0.808	0.662	0.583	0.546	0.527	0.503	0.476	0.462	0.456	0.423	0.349	0.323	0.304	0.283	0.255	0.252	0.003
flop/byte	**0.101**	**0.111**	**0.125**	**0.152**	**0.173**	**0.184**	**0.191**	**0.200**	**0.211**	**0.218**	**0.221**	**0.238**	**0.288**	**0.311**	**0.331**	**0.356**	**0.394**	**0.399**	**32**

Table 6: advect_scalar_pd: Effects of the cache

cache	0	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
loads	48.8	47.6	47.6	47.6	43.6	37.0	34.5	33.6	33.5	33.3	33.3	31.8	28.6	23.7	22.5	21.8	20.3	16.6	15.9
stores	6.6	6.6	6.0	6.0	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.3	5.3
all	55.4	54.2	53.6	53.6	49.0	42.3	39.8	39.0	38.8	38.7	38.7	37.2	34.0	29.1	27.9	27.2	25.6	21.9	21.2
load rate	1.000	0.976	0.976	0.975	0.893	0.757	0.706	0.689	0.686	0.683	0.682	0.652	0.586	0.485	0.461	0.446	0.415	0.339	0.326
store rate	1.000	1.000	0.909	0.909	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.816	0.816	0.816	0.815	0.813	0.810
all rate	1.000	0.979	0.968	0.967	0.884	0.765	0.719	0.704	0.701	0.699	0.698	0.672	0.613	0.525	0.503	0.490	0.463	0.396	0.383
flop/byte	**0.253**	**0.259**	**0.262**	**0.262**	**0.286**	**0.331**	**0.352**	**0.359**	**0.361**	**0.362**	**0.362**	**0.377**	**0.413**	**0.483**	**0.503**	**0.516**	**0.547**	**0.639**	**0.660**

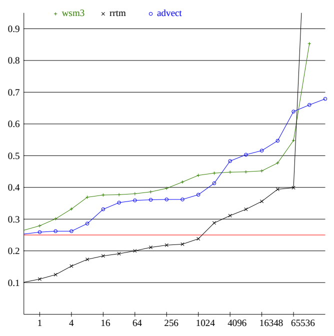

Fig. 2: The Arithmetic Intensity as a Function of the Cache Size

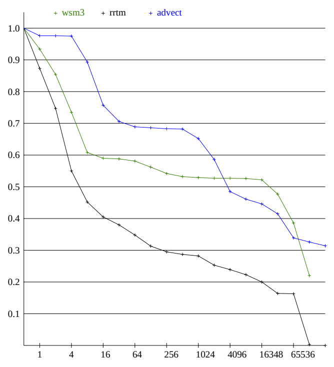

Fig. 3: The Memory Traffic as a Function of the Cache Size

References

1. https://www.hpcg-benchmark.org
2. https://www.top500.org