Parallel GNU APL

Jürgen Sauermann, GNU APL

Table of Contents

Abstract

This document briefly describes parallel execution in GNU APL

Configuration

Parallel execution of APL primitives needs to be **./configure**'d explicitly. Without such configuration GNU APL executes sequentially. In freshly installed GNU APL sources (i.e. after an SVN checkout or after unpacking the GNU APL tar file), the configuration is:

```
./configure --enable-maintainer-mode            \
            VALUE_CHECK_WANTED=no               \
            VALUE_HISTORY_WANTED=no             \
            PERFORMANCE_COUNTERS_WANTED=yes     \
            DYNAMIC_LOG_WANTED=yes              \
            ASSERT_LEVEL_WANTED=0               \
            CORE_COUNT_WANTED=-3
```

If the sources were already **./configure**'d then two make targets achieve the same:

```
make parallel
make parallel1
```

The first make target **parallel** is the setting for maximum performance, while the second make target **parallel1** is like **parallel** but with internal performance counters enabled. The performance counters are needed for benchmarking of the GNU APL performance via **FIO∆get_statistics** in workspace **5 FILE_IO** and **⎕FIO[201]** respectively.

Parallel execution is an entirely experimental feature that should not be used in mission-critical applications. Support for bugs caused by this feature is rather limited.

Controlling the Core Count

The number of cores that are used in parallel APL execution can be controlled in several ways.

The ./configure Option CORE_COUNT_WANTED

The first way to control the (maximum) core count is at the point in time when the interpreter source code is being ./configure'd. The configure option CORE_COUNT_WANTED defines how the interpreter chooses the number of cores (see **README-2-configure**):

- CORE_COUNT_WANTED=0: sequential
- CORE_COUNT_WANTED=1: parallel (but using only 1 core)
- CORE_COUNT_WANTED=2: parallel on 2 cores
- …
- CORE_COUNT_WANTED=N: parallel on N cores
- CORE_COUNT_WANTED=-1: parallel on all existing cores
- CORE_COUNT_WANTED=-2: parallel; the core count is determined by the command line argument -cc of the interpreter
- CORE_COUNT_WANTED=-3: parallel; the core count is determined at interpreter run-time via ⎕SYL;

The command line option -cc works only if CORE_COUNT_WANTED=-2. For CORE_COUNT_WANTED > 1 or -2, it is the responsibility of the user to ensure that the desired number of cores actually exist (i.e. this is not checked because some platforms cannot do that).

CORE_COUNT_WANTED=-3 is the most useful setting, because it allows changing the core count at run-time of an APL program (e.g. in a benchmarking workspace like **Scalar2.apl**).

Changing the Core Count at Run-time

If **CORE_COUNT_WANTED=-3** (the default) then the system variable **⎕SYL**, in particular **⎕SYL[24 25 26;]** control the number of cores being used:

```
      ⎕SYL[24 25 26;]
 CORE_COUNT_WANTED (per ./configure) ¯3
 cores available                     12
 cores used                           1
```

⎕SYL[24;2]

⎕SYL[24;2]) is a read-only value pertaining to the value used for **CORE_COUNT_WANTED** in **./configure**.

⎕SYL[25;2]

⎕SYL[25;2]) is the number of cores either chosen by ⎕SYL[24;2] or else as indicated by the platform. The value 12 above as reported on a i7-8700 Intel CPU with 6 physical cores and 2 core threads per physical core.

⎕SYL[26;2]

⎕SYL[26;2]) is the number of cores used by the interpreter. Setting:

⎕SYL[26;2])←0 chooses sequential operation of the interpreter. **⎕SYL[26;2])←N** with **N≥1 chooses parallel operation on *N** cores. * Other values trigger a

DOMAIN ERROR.

⎕SYL[26;2])←1 could be useful for benchmarking to see the difference between parallel and sequential code, but without distributing the computational load over several cores.

Setting **⎕SYL[26;2]** with a proper value in APL calls **Parallel::set_core_count()** in **Parallel.cc**.

The Initialization of the Parallel Subsystem

Parallel execution is initialized as follows:

- the initialization is performed automatically when the interpreter is started (function **Parallel::init()**). If the interpreter is ./configured with **DYNAMIC_LOG_WANTED=yes** then logging of the initialization can be enabled with command line option **-l 41**.
- **Parallel::init()** initializes semaphores related to parallel execution and then calls **CPU_pool::init()**. **CPU_pool::init()** determines, which CPUs can be used by the interpreter and stores them in its vector **the_CPUs**. If **CORE_COUNT_WANTED** is **≥ 0** or **-2** then the CPUs in the vector are determined by **CORE_COUNT_WANTED** or by the **-cc** command line option (no checks are performed to verify that the CPUs chosen are correct). Otherwise, i.e. (**CORE_COUNT_WANTED** is -1 or -3) the cores available to the interpreter are determined by **pthread_getaffinity_np()** and all CPUs that are available to the interpreter are stored in the vector.
- Then **Parallel::init()** creates a thread pool with **Thread_context::init_parallel**, with one thread for each CPU in **CPU_pool::the_CPUs**. If **CORE_COUNT_WANTED = -3** then only the first thread is activated (and the user needs to use ⎕SYL in order to activate more cores. Otherwise all threads are activated (and ⎕SYL cannot be used). Finally **Parallel::init()** brings all threads into their initial state.

At any point in time, a thread can be in one of 2 states:

- BLKD: Blocked on its private semaphore **Thread_context::pool_sema**, or
- RUN: Running.

A thread in state RUN can further be in 2 sub-states:

- busy-waiting for more work to become available, or
- computing the current job.

The first thread in the pool, aka. the **master**, is always in state **RUN** and is never busy-waiting (instead it executes the APL interpreter).

The remaining threads, aka. the **workers**, are in state **BLKD** as long as they are inactive (this can only happen if ⎕SYL is being used and the worker is above the value set with ⎕SYL). Otherwise the worker is in state **RUN**. A worker in state **RUN** is not necessarily computing, e.g. if the joblists are empty and the worker is busy-waiting for more work.

When the interpreter (i.e. the master) needs to compute a primitive scalar function (or an inner or outer product of a primitive scalar function) with a sufficiently large argument, then it unleashes the workers (**Thread_context::M_fork()**), performs its own share of the work, and waits for all workers to complete their share of the work (**Thread_context::M_join()**).

At the same time, the workers wait for the master's **M_fork()** in **Thread_context::PF_fork()**, perform their share of the work, indicate that their work is complete, and wait for all others to complete as well (**Thread_context::M_join()**).

The Operation of the Parallel Subsystem

After initialization, the parallel subsystem works (see **Thread_context.cc/hh**) as follows.

- worker thread in state BLKD do nothing. This case can only occur with **CORE_COUNT_WANTED=-3**, and the transition between states BLKD and RUN can (after the initialization) only occur by setting ⎕SYL[26;2].
- every thread maintains a variable **job_number** which is initially 0 for both the master and every worker:

```
Thread_context::Thread_context()
   : N(CNUM_INVALID),
     thread(0),
     *job_number(0)*,
     job_name("no-job-name"),
     blocked(false)
{
}
```

- As long as the **Thread_context::job_number** of the master is equal to the **Thread_context::job_number** number of a worker, that worker busy-waits until both numbers differ. The master also increments the static variable **busy_worker_count**:

```
/// start parallel execution of work in a worker
void PF_fork()
   {
     while (get_master().job_number == job_number)
           /* busy wait until the master has increased job_number */ ;
   }
```

- When the master finds new work (e.g. after interpreting a scalar APL function) then it inserts that work into the proper **Parallel_job_list<>** of each worker and increments its own **Thread_context::job_number** (in **Thread_context::M_fork()**). This causes all workers to begin their share of the work:

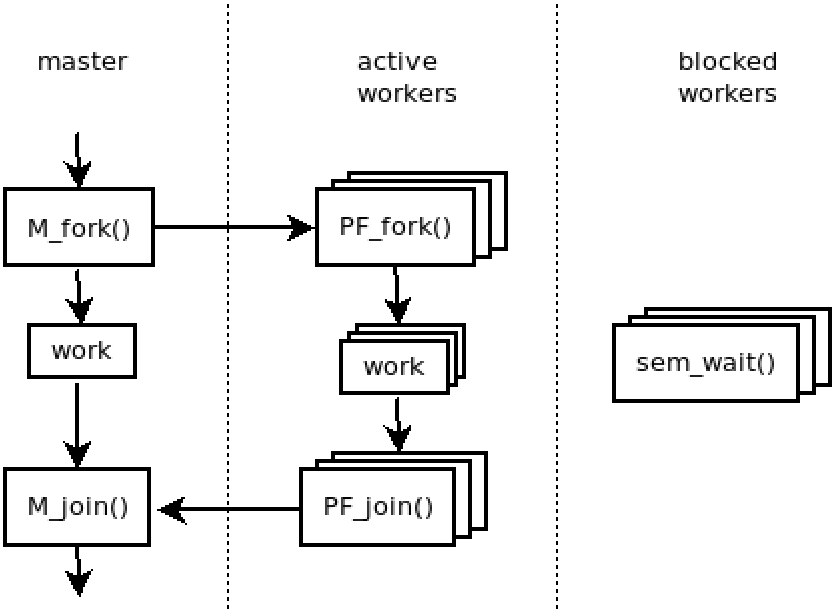
```
/// start parallel execution of work at the master
static void M_fork(const char * jname)
   {
     get_master().job_name = jname;
     atomic_add(busy_worker_count, active_core_count - 1);
```

```
      ++get_master().job_number;
    }
```

- The workers perform their work and, after finishing it, increment their **Thread_context::job_number** and decrement **busy_worker_count**.again:

```
/// end parallel execution of work in a worker
void PF_join()
   {
     atomic_add(busy_worker_count, -1);   // we are ready
     ++job_number;            // we reached master job_number

     // wait until all workers finished or new job from master
     while (atomic_read(busy_worker_count) != 0 &&
            get_master().job_number == job_number)
           /* busy wait */ ;
   }
```



The synchronization scheme above was designed such that as little interaction between threads is needed and heavier constructs like semaphores could be avoided.

Notation

In the context of parallel execution, the prefix **M_** designates functions that are only called from the master thread, while the prefix **PF_** (for pool function) designates functions that are called from a worker thread.

Master functions only exist in class **Thread_context**, while pool functions exist in classes **Thread_context**, **ScalarFunction**, **Bif_OPER2_INNER**, and **Bif_OPER2_OUTER**. Note that the master thread itself acts like a worker thread after returning from **M_fork()** and before calling **M_join()**.

Benchmarking of the Parallel Execution

The Theory …

If a scalar APL function, is computed on a single core, then the time (most conveniently expressed in terms of CPU cycles) to compute it for an APL array with a ravel of length N is:

$T_{seq}(N) = \alpha_{seq} + \beta_{seq} \times N$.

In theory, the parallel computation of the same function on a number of cores requires time:

$T_{par}(N) = \alpha_{par} + \beta_{par} \times N$.

The terms α_{seq} and α_{par} are the start-up times for the computation, while the terms β_{seq} and β_{par} are the per-item times for the computation.

Under normal circumstance one has:

- $\alpha_{seq} \le \alpha_{par}$
- $\beta_{seq} \ge \beta_{par}$

Under ideal circumstances one even has

$\beta_{par} = \beta_{seq} \div C$, or: $\beta_{seq} \div \beta_{par} = C$.

where C is the number of cores involved. The quotient **$\beta_{seq} \div \beta_{par}$** is commonly known as the **speed-up** of the parallel execution. The difference α_{par} - α_{seq} is primarily caused by functions like M_fork(), PF_fork(), M_join() and PF_join() above, but also by the overhead caused by the joblist mechanism that is required to efficiently parallelize scalar operation on nested APL values.

The equations above can be used to compute a break-even length N_{BE} so that:

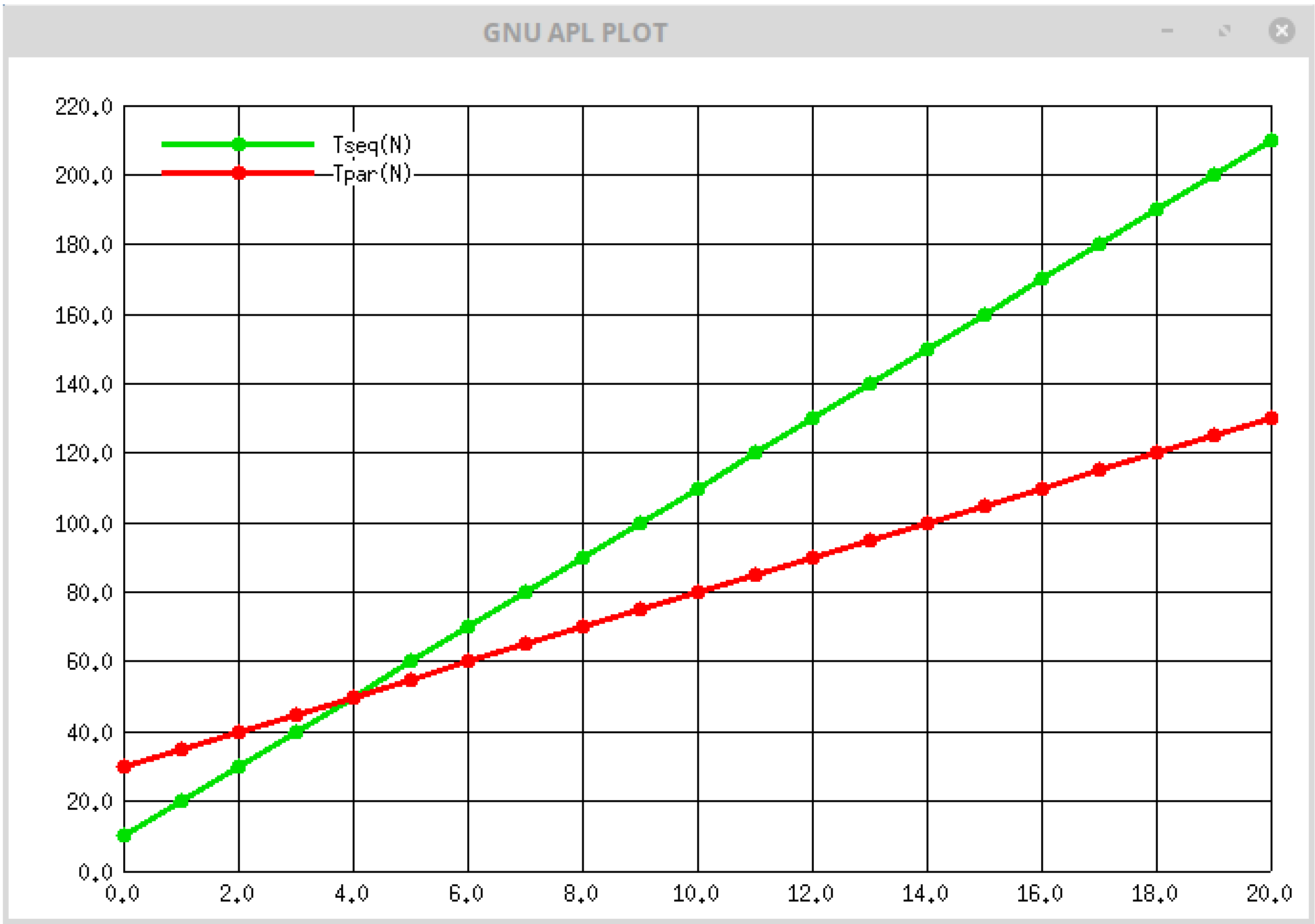
- $T_{seq}(N) < T_{par}(N)$ for $N < N_{BE}$
- $T_{seq}(N) > T_{par}(N)$ for $N > N_{BE}$.

That simply means that the computation for arrays with a short ravel (i.e. of less than N_{BE} items) it is faster to compute sequentially, while for longer ravels it is faster to compute in parallel.

The above formulae are easier to interpret if one plots the execution times (on the Y axis) vs. the vector length (on the X axis). For example, if

- $\alpha_{seq} = 10$, $\alpha_{par} = 10$, i.e. $T_{seq}(N) = 10 + 10 \times N$ (green plot line)
- $\beta_{seq} = 30$, $\beta_{par} = 5$, i.e. $T_{par}(N) = 30 + 5$ (red plot line)

then the theory predicts the following execution times:



As one can see, the intersection of the Y-axis (i.e. N=0) and the plot line $T_{seq}(N)$ and $T_{par}(N)$ is the start-up time α_{seq} and α_{par} respectively. The break-even length in this example is the intersection of the two plot lines at N=4.

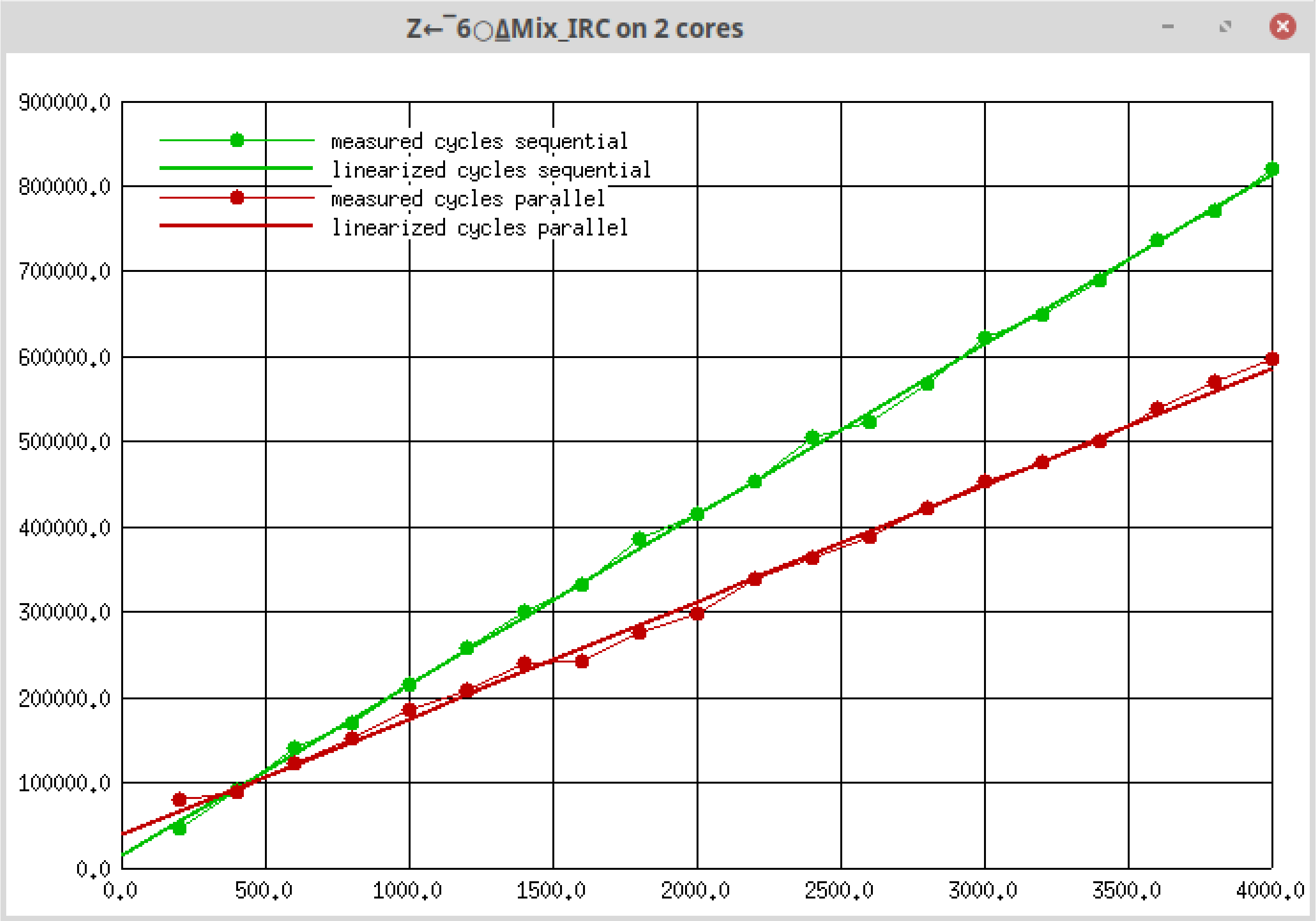
… and the Practice

As **Benjamin Brewster** stated in 1882: *In theory there is no difference between theory and practice, while in practice there is*.

This statement is particularly true for benchmarking. Until about 1990, given some piece of assembler code, it was feasible (and was actually done) to compute the number of CPU cycles that the execution of that code would take.

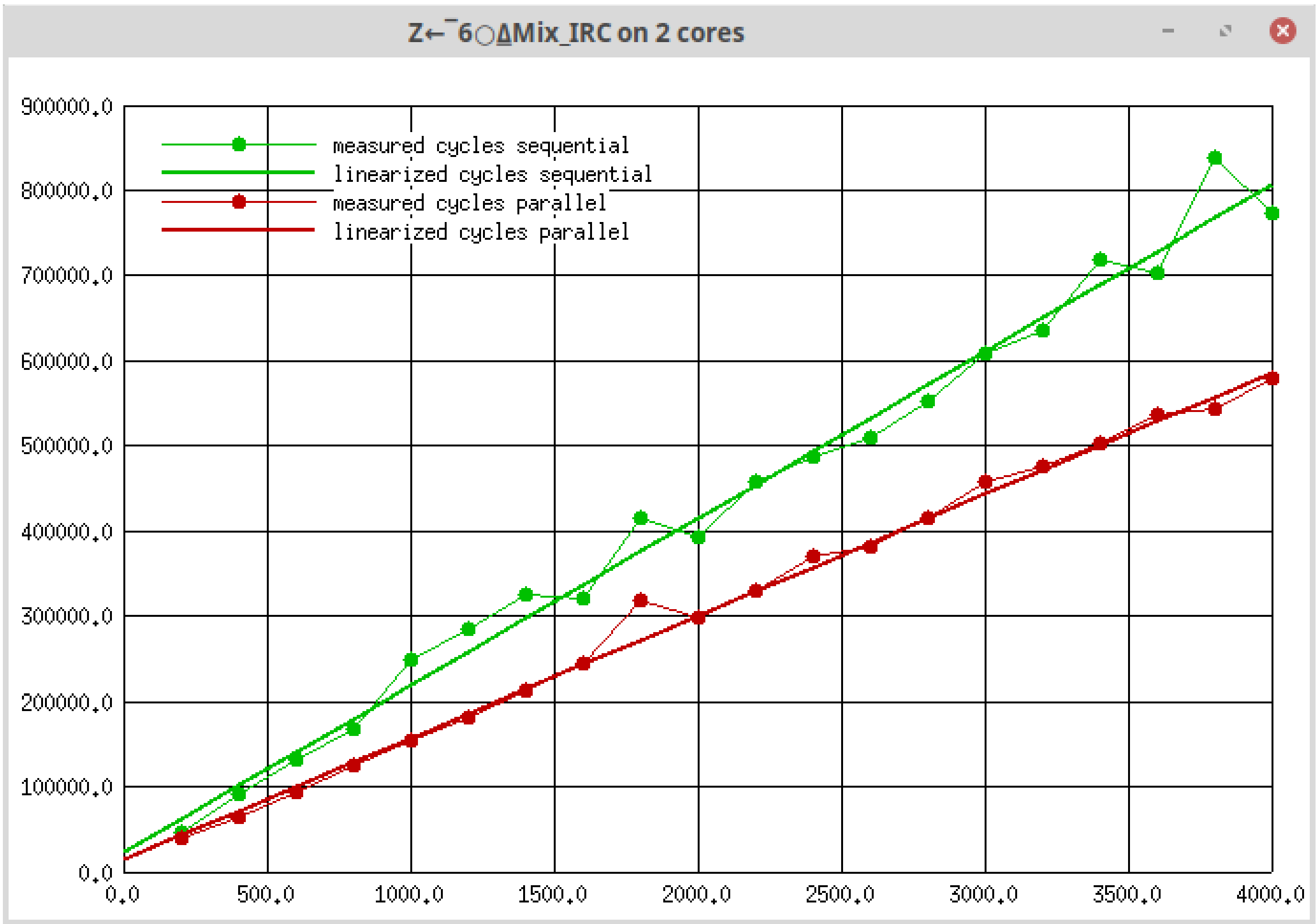
Since then a number of optimizations, both in hardware and in software, have made it practically impossible to predict the execution time of any given code. Even worse, these days the same code, executed again and again, typically results in rather different cycle counts for each execution pass. Even if "no" other processes execute on the same CPU on which a benchmark measurement is performed (where "no other process" means not counting the typically 250 or so operating system processes that are sitting idle on the CPU) the results can differ substantially between different measurements of the same code.

So in practice, lets discuss the results of a benchmark:



This benchmark measured the time to compute Z←¯6 ○ MixIRC for different vector lengths, ranging from N=200 to N=4000. MixIRC is a random mix of integer, real and complex arguments of ¯6○ aka. **arccosh**. The benchmark worked well in the sense that the measured numbers of CPU cycles were very much in line with the theory. The thick lines are those that have the smallest squared differences from the measurement points (the line that best matches the measurement points).

To be on the safe side, lets repeat **the same** benchmark:



This one went less well. One difference from the previous one is that the deviations of the measurement points are considerably larger than in the previous run. If one runs the benchmark many times, then it looks like the deviations in the sequential execution are larger than in the parallel execution. More importantly, the sequential start-up time α_{seq} is now larger than the parallel start-up time α_{par}.

These two examples are only meant to highlight some the problems that may occur if one tries to determine the parameters α and β. The following is a summary of findings after having performed many such measurements with GNU APL and different core counts, vector lengths, and primitive functions:

- every measurement needs to be visualized (plotted) to rule out too many or too large outliers.
- for determining the start-up costs α_{seq} and α_{par} it seems to be better to use fewer vector lengths and also shorter vectors.
- for determining the per-item costs β_{seq} and β_{par} it is better to use longer vectors.
- scalar functions with a low β (like A+B) tend to give more obscure results (and lower speed-ups) than scalar functions with a higher β. This is primarily caused by the fact that all cores share the same interface to the (shared) main memory of the machine.
- The speed-up of additional virtual cores (compared to physical ones) seems to be rather low. That is, for example, the speed-up of 12 virtual cores (on a hyper-threaded CPU with 6 physical cores) is only marginally higher than on 6 physical cores. GNU APL addresses this fact by distributing the load over

the physical cores before placing hyper-threads on the physical cores.

The Benchmark Workspace Scalar2.apl

The workspace **workspaces/Scalar2.apl** can be used to measure the execution times of scalar functions. GNU APL provides a number of internal performance counters. These counters need to be enabled with **PERFORMANCE_COUNTERS_WANTED=yes** in **./configure**, and the CPU must have a cycle counter and an instruction to read it (currently only Intel CPUs can use this feature). The cycle counter of the CPU is read before and after the computation of a scalar function, and the difference can be read in APL via **⎕FIO[200]** and **⎕FIO[201]**. Measuring execution times this way is far more precise than old-fashioned measurements using ⎕TS at the APL level.

Scalar2.apl is most conveniently called from the command line, and what is being measured can be controlled via command line arguments. For example (from the top-level directory of GNU APL):

```
make parallel1      # runs ./configure with suitable options
src/apl -f workspaces/Scalar2.apl -- -c 3,6 -d 200×⍳20
```

The **Scalar2.apl** workspace understands the following command line options:

Table 1. Table Scalar2.apl command line options (after --)

Option	**Effect**	**Example**	**Default**
-c core-counts	set the number of cores	-c 2,3	2
-d vector-lengths	set the vector lengths (N-axis)	-d 200×⍳2	⍳20
-f function	select the function to measure	-f 20	39

For every core count, **Scalar2.apl** displays a separate plot window with the measurement results for sequential execution and for the parallel execution with the given core count.

Recursive Parallelization

The purpose of the joblist mentioned above is as follows. Consider the APL expression below, computed in parallel on 4 cores:

```
Z←1 2 (⍳1000) 4 + 1 (20 21 22) 3 4
```

The 4 ravel elements of the left and right arguments of dyadic + are stored in 4 consecutive Cells, which are distributed in a round-robin fashion over the cores. That is:

```
Core #1 computes: 1 + 1              (1 addition)
Core #2 computes: 2 + 20 21 22       (3 additions)
Core #3 computes: (ι1000) + 3        (1000 additions)
Core #4 computes: 4 + 4              (1 addition)
```

Therefore cores #1 and #4 compute one sum, core #2 computes 3 sums, and core #3 computes 1000 sums. This is obviously not optimal since cores #1, #2, and #3 are most of the time idle, waiting for core #3 to finish.

To avoid this case, GNU APL parallelizes scalar functions recursively with the following algorithm.

1. the interpreter starts with an empty joblist.
2. when the interpreter evaluates a scalar function, then it puts a new job into the joblist. The job describes the relevant parameters (essentially the scalar function to be computed and the address(es) of its argument(s).
3. LOOP: while the joblist is not empty:
    1. remove the first job from the list
    2. perform the computation defined in the job in parallel
    3. if a core comes across a nested ravel item, then:
        - if the item (and hence the result) is small: compute it immediately
        - if the item is large: create a new APL value whose ravel is uninitialized (this operation takes constant time) and add a new entry into the joblist (for computing the ravel of the nested result later on).

For performance reasons, there are actually two such joblists: **Thread_context::joblist_B** for monadic scalar functions, and **Thread_context::joblist_AB** for dyadic scalar functions (and inner and outer products of them).

Setting Thresholds

One purpose of benchmarking is to find the break-even lengths for scalar functions. After that length is found, one can inform the APL interpreter about the break-even lengths. This is done via a configuration file, normally **/usr/local/etc/gnu-apl.d/parallel_thresholds**.

This file is installed by *make install*, but the values in the file are usually not optimal. One can, however, enter better values manually. Consider a few non-empty lines in the file:

```
perfo_1(F12_PLUS,      _B,   "+ B",    8888888888888888888ULL)
perfo_1(F12_POWER,     _B,   "⋆ B",    12                    )
```

```
perfo_2(F12_TIMES,     _AB,  "A × B",  33                    )
```

The first line above sets the break-even point of **monadic +** to 8888888888888888888ULL, which is a value so large that parallel execution will never happen for monadic +.

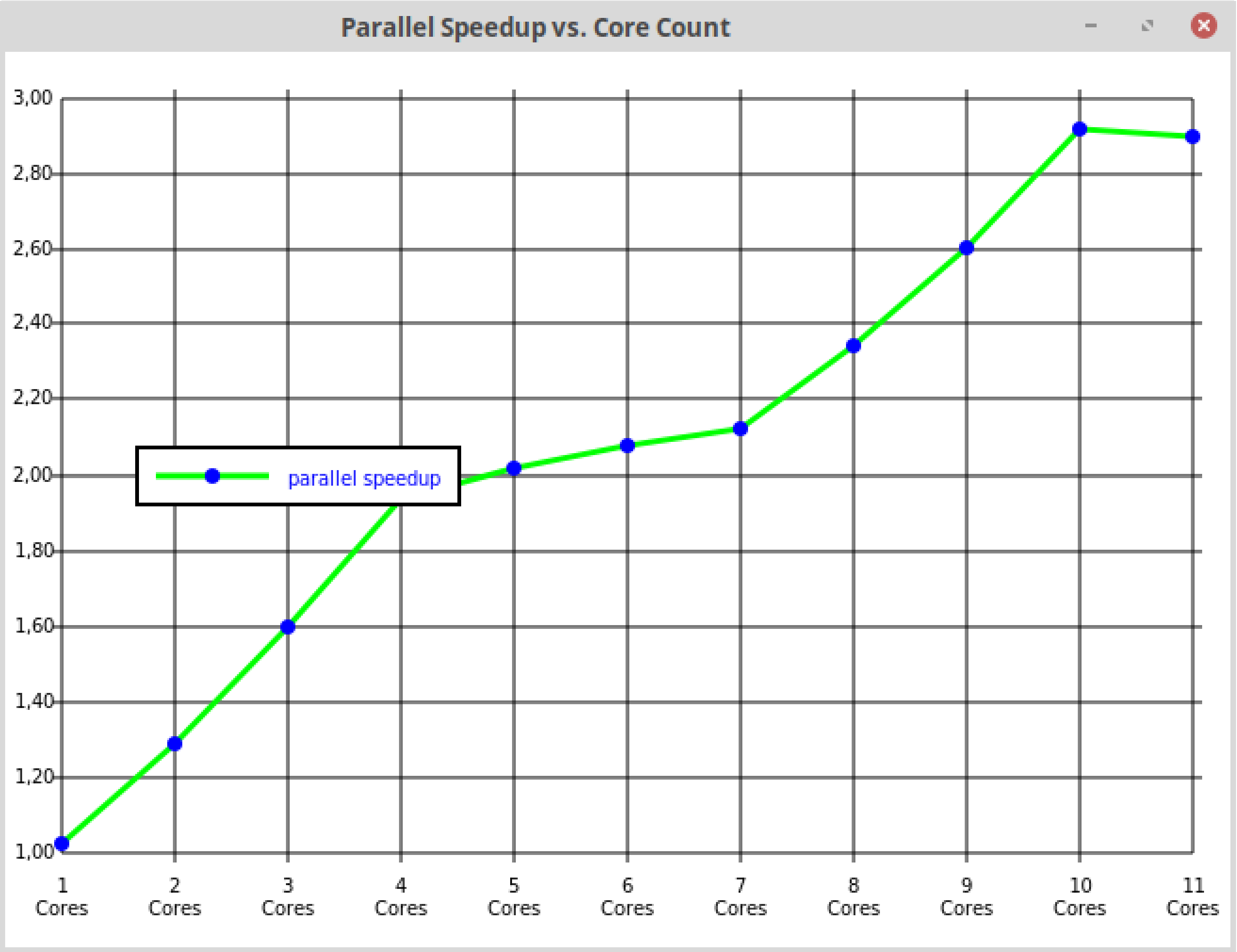
The second line sets the break-even point of **monadic *** to 12. Arrays (of any rank) with fewer than 12 ravel items will be computed sequentially, but longer arrays in parallel.

The third line sets the break-even point of **dyadic ×** to 33. Arrays (of any rank) with fewer than 33 ravel items will be computed sequentially, but longer arrays in parallel.

In general, the fewer cycles a function needs, the higher should the threshold be set.

The Optimal vs. the Maximal Core Count

The last plot window shown by the **Scalar2** workspace summarizes the speed-ups that were achieved for the different core counts that were selected with the -c option.



leave at least one of the available cores unused because otherwise the operating system could grab one of the cores for its own purposes and that core would then become much slower than the others. The operating system is usually smart enough to locate an unused core for its own purposes, but that must fail if all cores are fully loaded with APL work.

The speed-ups plotted in the last window are usually reasonable if:

- the APL function that is being bench-marked (as selected by the -f option) is not too lightweight, and
- the vector lengths (-d option) are not too short.

The measurement shown above was executed on a 6-core i7-8700 CPU. The CPU has 6 (physical) cores with 12 (logical) core threads where the 12 core threads are mapped to 12 "CPUs" in the operating system as (per **/proc/cpuinfo**).

The benchmark command used was:

```
src/apl -f workspaces/Scalar2.apl -- -c ι11 -d 5000×ι20 -f 39
```

which means that:

- **-f 39** : selects **Z←¯6○B** aka. **arccosh(B)** as the benchmarking function with a mix of Integer, Real, and Complex numbers,
- **-d 5000×ι20** : use vector lengths ⍴B = 5,000, 10,000, 15,000 ... 100,000, and
- **-c ι11** : use core counts of 1, 2, 3, ... and 11.

The 11 different core counts and 20 different vector lengths above amounted to a total of 220 (= 11×20) individual measurements in the benchmark plotted above.

Had we run the benchmark with **-c ι12** instead of **-c ι11** then the measurement with 12 cores would have shown a rather bad speed-up (try it yourself with **-c** set to the number of cores on your computer).

Corollary: the optimal performance is achieved when the number of cores used for parallel APL is slightly smaller than the number of cores available. The number of cores available is also influenced by other (even idle) processes on the machine because they also influence the scheduling of the threads that perform the APL work.

Last updated 2024-03-14 14:48:59 CET