Quick-n-dirty ways to run your serial code faster, in parallel

Sergey Mashchenko
McMaster University
(SHARCNET)

Outline

- Foreword
- Parallel computing primer
- Running your code in parallel
  - Serial farming
  - Using pagecache to accelerate I/O bound code
  - Automatic code parallelization
  - Semi-automatic parallelization: OpenMP
  - Semi-automatic parallelization: OpenACC
- Summary

Foreword

- This talk is not about optimizing / profiling a serial code (compiler optimization flags etc.)
- Instead, this talk is about accelerating the computations by running your serial code in parallel.
- Only the simplest parallelization techniques are considered; as a result, this will work well only for some codes and problems.

Parallel computing primer

What is a parallel code?

- A code is parallel when it consists of multiple processes running at the same time on multiple compute resources (CPU or GPU cores).
- Optionally there may be a need for data exchange (“communication”) between the processes, often requiring synchronization between processes.
  - When no data exchange is needed, we have an “embarrassingly parallel” case – or serial farming

Memory

- Memory-wise, two situations exist:
  - When every process can access any byte of the memory (“global address space”, “global memory”), we have a “shared memory” situation.
    - On a single cluster node
    - “Device memory” on GPU
  - If this is not the case, we have a “distributed memory” situation.
    - Between cluster nodes
    - Between CPU and GPU

Programming models

- At a low level, three basic programming models are in common use:
  - Distributed memory model (MPI)
  - Shared memory model (threads)
  - GPU model (CUDA/OpenCL) – actually a special case; a combination of distributed and shared memory models (+ vector computing)
- In the rest of the talk, only higher level approaches will be discussed; they are using the above low level models under the hood.

Is parallel computing hard?

- There are many myths regarding parallel computing, e.g.:
  - Only “hard core” parallelization (converting a serial code to MPI / CUDA / pthreads) is the true one
    - Much simpler approaches considered here result in “true” parallel codes, albeit less efficient in some cases
  - It takes many months or even years to parallelize a large serial code
    - With the approaches considered here a conversion would probably take less than a day

Caveats

- In a shared supercomputing resources environment (SHARCNET), running your code in parallel needs to be justified
  - It takes longer to wait for N cores than to wait for a single core, so if the speedup is not great, the total time (queuing + running) can become larger for the parallel version of the code.
- Also, very low parallel code efficiency is a waste of resources
  - A rule of thumb: running on N CPU cores, the speedup should be at least $0.7*N$
  - For GPUs, the speedup should be at least ~10x.

Theory

- Using a parallel code on the same size problem as the serial code usually results in low efficiency (speedup)
  - So-called Amdahl's law, or “strong scaling”
- The solution: use the parallel code on a larger size problem (more grid elements; more particles; more Monte Carlo steps; etc.)
  - Gustafson's law (“weak scaling”)
- Serial farming does not suffer from these issues

Running your code in parallel

Serial farming

Definition

- Serial farming: running multiple copies of a serial code on multiple CPU cores at the same time.
- In the simplest case, there are no data dependencies
  - Meaning the final result does not depend on the order of execution of the serial jobs.
- In more complex cases, there may be dependencies between groups of serial jobs.
  - E.g., don't start group 2 until all the jobs in group 1 are finished.
  - This can be handled by using inter-job dependency features of the scheduler:

```
sqsub --waitfor=jobid[,jobid...]
```

Myths vs. reality

- “Serial farming is not parallel computing”: myth. See the definition of a parallel code.
- “One should avoid using serial farming, because it is embarrassingly parallel”: myth.
  - Due to zero overhead (no communications), serial farming should rather be called “perfectly parallel”.
    - That is, when running on N cpu cores, the speedup is N (100% efficiency).
  - Also, the queuing time for say 128 serial farm jobs is much shorter on average than for a “true” 128-way parallel job.

Typical applications

- Monte-Carlo type simulations
  - There is an implicit data dependency here: to make sure that all serial jobs are using unique random number sequences.
  - This is not an impediment, and can be easily handled (see the tutorial “Serial farming and Monte Carlo for SHARCNET” on SHARCNET's Help Wiki).
- Model parameter study
  - Model input parameters are often not precisely known, so one has to run a set of simulations with the parameters varying within the acceptable range.

Implementation

- One can use any scripting language (bash, perl, python, ...) to write serial farm scripts – for job submission, queries, killing, post-processing.
- This bash script handles the common situation when a serial code has to run with a set of parameters, stored one line per job in a file:

```
while read line
do
  sqsub  -o out%J  -r 7d  ./code "$line" | cut -d" " -f4 >> jobid
done  < input_parameters.dat
```

Then the whole job batch can be killed with

```
sqkill  `cat jobid`
```

Running your code in parallel

Using pagecache to accelerate I/O bound code

Definition

- I/O bound code is the one where most of run time is spent in reading from and/or writing to the disk.
  - Such codes waste lots of CPU cycles, and can overload our file systems (making it very slow for everyone).
  - This is especially true if these jobs are run as a serial farm, on random nodes.
- But: if multiple processes read the same data and run on the same node, the Linux feature pagecache can dramatically accelerate computations.
- No changes to the code needed!

Pagecache

- Pagecache is the cache of recent reads and writes, occupying all the unused memory in a node. It is operated by the Linux kernel.
- As long as there is enough of unused RAM to fit all the data which are being read, the reading from the disk only occurs once; all other code instances will get the data from the memory cache, which is dramatically faster.
- Serial farm using the pagecache feature is essentially a single multi-threaded parallel application, so one has to use the threaded queue to submit such jobs, e.g.:

```
sqsub -q threaded -n 24  -o out  ./job_script.sh
```

Here job_script.sh is a script launching 24 instances of the serial code (see next slide).

Details

-
  ```
  for ((i=0; i<24; i++)); do ./code [args] &; done
  wait
  ```
- Caveat: asking for many cores on a single node can result in a substantial queue wait time
  - True for any multi-threaded application
- For a code reading extremely large amounts of data, iqaluk is the best system to use
  - Only 32 cores, but 1TB of RAM; currently no scheduler
- Success story: one group (climate modelling; 50,000 serial jobs, each one reading 1 TB of the same data) got their results 25x faster, by switching from serial farm on random orca nodes to using iqaluk.

Running your code in parallel

Automatic code parallelization

Compiler based parallelization

- Modern compilers can optionally compile your serial code as a parallel (multi-threaded) code, in a fully automatic fashion.
- Specifically, our intel compilers need -parallel option to carry out auto-parallelization:

```
icc -O3 -parallel code.c
ifort -O3 -parallel code.f90
```

- Code compiled in this fashion is a “true” multi-threaded application, and needs to be submitted to the threaded queue, e.g.:

```
sqsub -q threaded -n 24 -o out -r 3d ./code
```

Details

- When running the code interactively (on a development node), one can choose the number of threads to use by assigning a value to this environment variable:

```
export OMP_NUM_THREADS=24
```

- Parallelization only targets loops, with no data dependencies, and the number of iteration known at compile time.
- -guide-par, used in conjunction with -parallel, causes the compiler to generate advisory messages suggesting ways the programmer might help the compiler to auto-parallelize suitable loops. No object file is generated.

Running your code in parallel

Semi-automatic parallelization: OpenMP

Preamble

- Fully automatic parallelization is too limited to be effective for most codes. Programmer can achieve significantly better results by guiding the parallelizing process.
- On CPU cores, the standard approach for semi-automatic parallelization is OpenMP.
- OpenMP does require some modifications to the code, but unlike “hard-core” approaches (MPI, pthreads, CUDA), OpenMP allows for *incremental parallelism* (the modified code still can be compiled and used as a serial code).

Simplest case

- OpenMP supports a wide range of parallel programming tools, but we will only consider the simplest – loop parallelization:

```
#pragma omp parallel for       // C language
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

!$omp parallel do     // Fortran
    do i = 1, N
         a(i) = 2 * i
         end do
```

Details

- By default, all variables in the work sharing region are shared except the loop iteration counter.
- Inside-loop variables which value depends on the loop index should be declared as private:

```
!$omp parallel do private(A)
        do i = 1, N
        A = 2 * i – 1
        C(i) = sqrt(A)+log10(A)-log(A)
        end do
```

Compiling / running

- If OpenMP code is compiled without -openmp switch, all OpenMP pragmas are ignored, and it is compiled as a serial code.
- If you add -openmp switch, a multi-threaded binary is generated. In intereactive use, the environment variable OMP_NUM_THREADS is used to specify the number of threads. When submitting to the scheduler, the threaded queue should be used:

```
sqsub -q threaded -n 24 -o out -r 3d ./code
```

Running your code in parallel

Semi-automatic parallelization: OpenACC

Introduction

- OpenACC is an equivalent of OpenMP, for GPU computing.
- SHARCNET has two GPU clusters:
  - angel (44 older GPUs; not good for double precision);
  - monk (108 newer GPUs; good for double precision)
- Only one compiler – PGI – has openACC support. To use it on angel/monk:

```
module unload intel
module load pgi
```

Simplest case

- As in OpenMP, the obvious targets for OpenACC are data-independent loops. Examples:

```
#pragma acc kernels loop        // C language
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

!$acc kernels loop     // Fortran
    do i = 1, N
        a(i) = 2 * i
        end do
```

Caveat

- The important difference from OpenMP: as loops which are being parallelized under OpenACC will have to copy all the input/output data between CPU and GPU via a relatively slow PCI-E link, the loop content has to be CPU-intensive (flop/byte ratio has to be high).
  - Bad flop/byte ratio:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];
```

  - Better flop/byte ratio:

```
for (i=0; i<N; i++)  {
    A = 2*i – 1.0;
    C[i] = sqrt(A) + log10(A) – log(A);  }
```

Compiling / running

- To compile:

```
pgcc -Minfo=accel -fast -v -acc code.c
pgf90 -Minfo=accel -fast -v -acc code.f
```

- To run interactively (on monk dev node, mon54):

```
./code
```

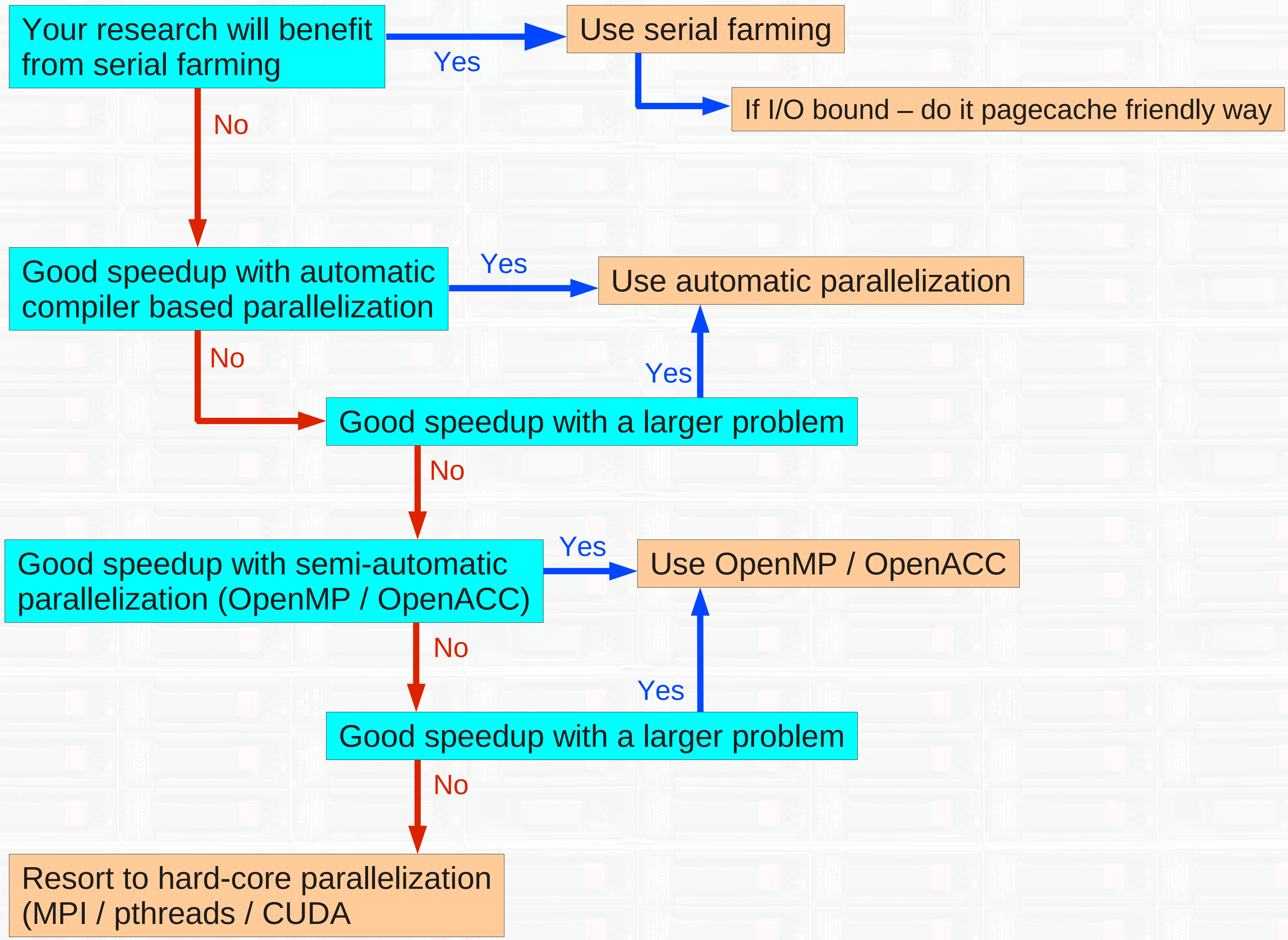
- To submit to the scheduler (angel, monk):

```
sqsub -q gpu -o out -r 1d  ./code
```

Summary

Final remarks

- With the exception of plain serial farming, all the considered parallelizing approaches result in a multi-threaded code, and hence can only be run on a single node (shared memory environment).
- The first two methods (serial farming with and without pagecache feature) require no source code, and can be utilized with a binary code (e.g. commercial).
- Only the last two methods (OpenMP and OpenACC) require some code modifications.


Your research will benefit from serial farming
Yes
Use serial farming
If I/O bound – do it pagecache friendly way
No
Good speedup with automatic compiler based parallelization
Yes
Use automatic parallelization
No
Yes
Good speedup with a larger problem
No
Good speedup with semi-automatic parallelization (OpenMP / OpenACC)
Yes
Use OpenMP / OpenACC
No
Yes
Good speedup with a larger problem
No
Resort to hard-core parallelization (MPI / pthreads / CUDA