Approaches towards faster computation

S. Yashonath
SSCU,
IISc,
Bangalore 560012

In collaboration with

Dr. T.S. Mohan, Infosys Technologies Ltd.

Dr. Raghu Hudli, Managing Director, ObjectOrb
Shrinidhi Hudli
Shrihari Hudli

How to make computation faster :

Improvement in methods – leads to substantial improvement in turnaround time

Algorithmic changes – also leads to significant but not substantial improvement (e.g., the methods for searching an ordered database – using the binary search)

Use a supercomputer -- as many computer centres do. But not possible for many or most users.

Use a parallel computer : uses many cores/CPUs to parallelize the problem.

Use a graphics processing unit.

Method Improvement :

Examples of this are

1)From simple hit or miss Monte Carlo to crude MC to importance sampling MC.
2)From Born-Oppenheimer approach to solving electronic degrees of freedom to Car-Parrinello approach.
3)From multidimensional integral to methods that lead to closest estimate such as molecular dynamics method.

Algorithimic changes :

Example :
1) Instead of normal search do a binary search of the database ($\log_2 N$) instead of order N.

Supercomputer :

They attempt to have faster processors and this makes them fast. These require special installation
facilities and are power intensive.
They cost a fortune and only a few can afford.

Parallel Computers :

Employs multiple CPUs or cores and in this comes SIMD and MIMD.

Single Instruction Multiple Data.
Multiple Instruction Multiple Data.

Parallel Programming

Motivation to parallelize long-running programs and achieve faster execution times

- Concurrently execute steps (parts) of a program

Search for parallelism is problem dependent

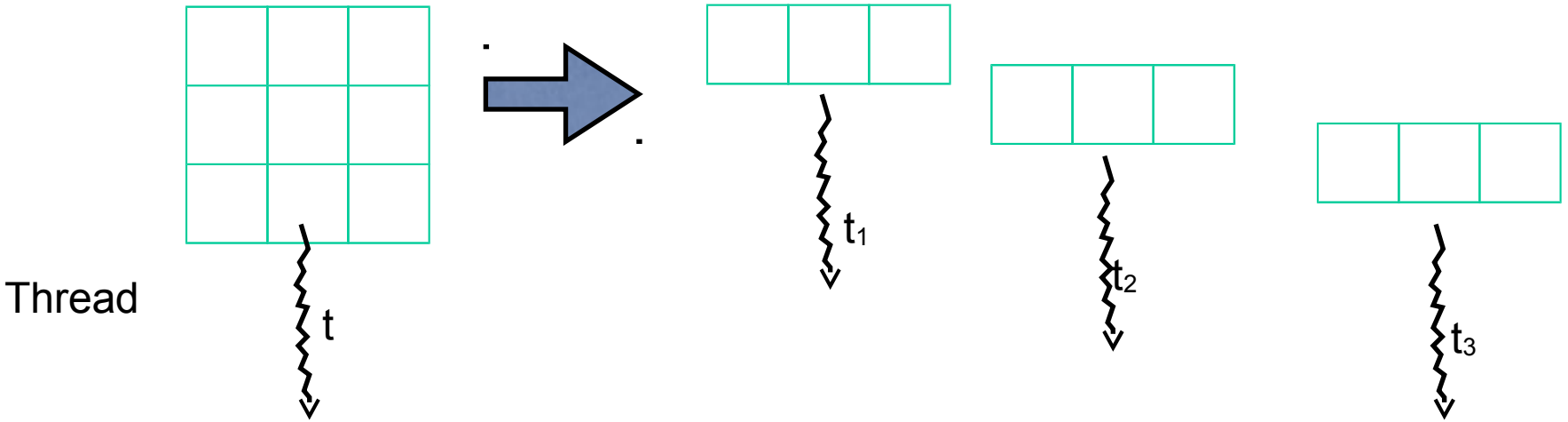
- Data Parallelism
- Task Parallelism
- Pipelining

Models of Parallelism

What do we do in parallel?

- Access/transform data in parallel but do the same task - data parallelism
- Work on multiple, but related tasks in parallel - task parallelism
- Different and independent tasks to be applied on a data stream - pipeline parallelism; combination of data and task parallelism

Data Parallelism

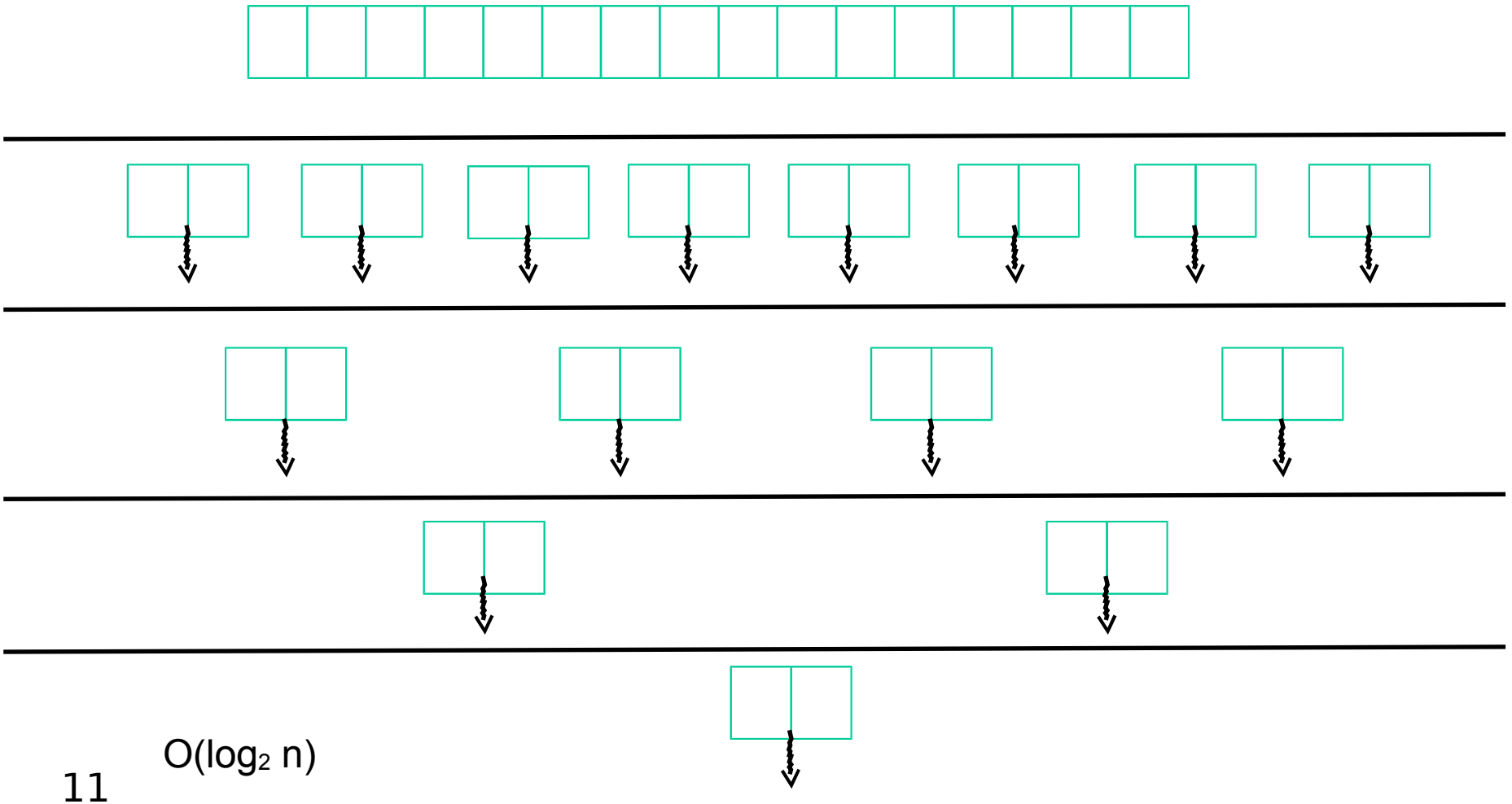


Each thread works on a mutually exclusive data set

Does “similar” work

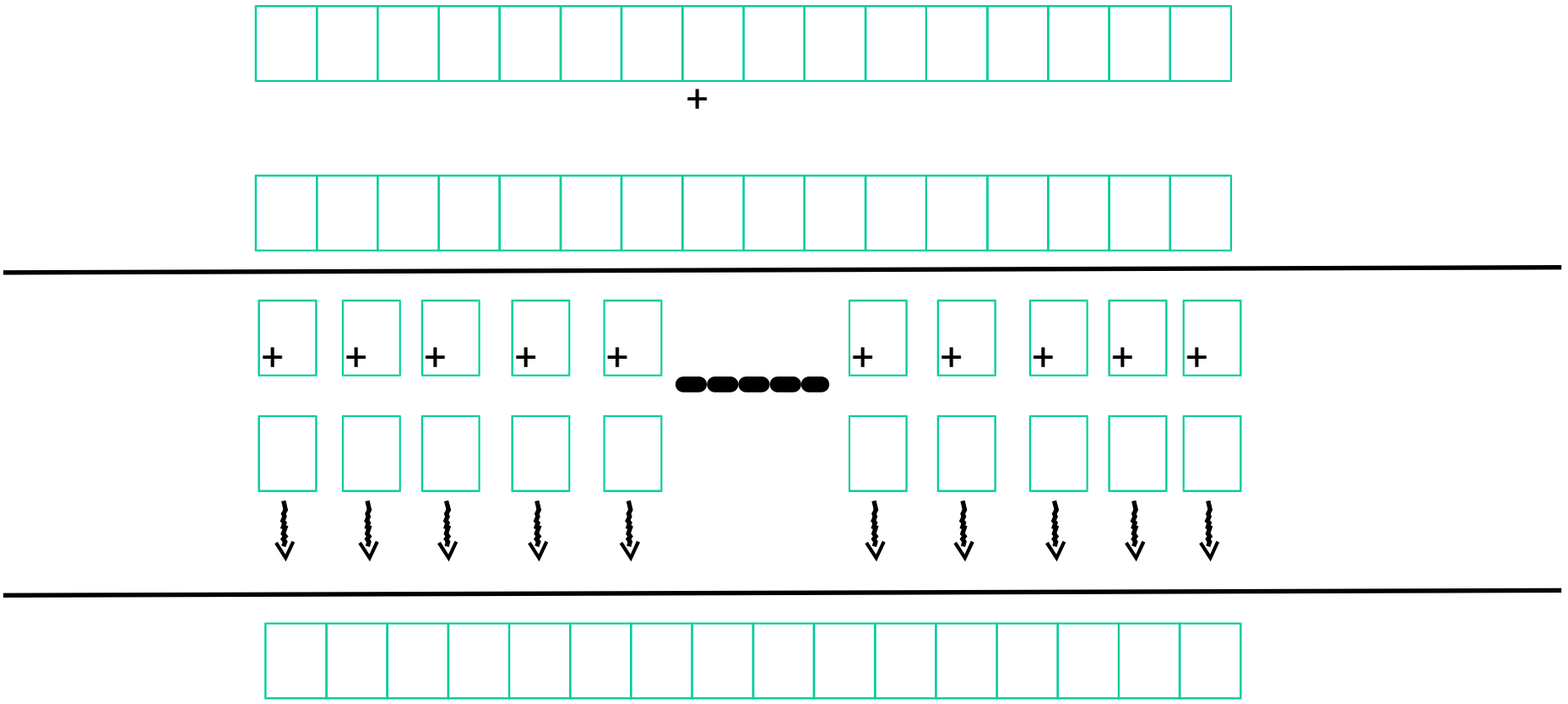
- Ideal case, requiring no synchronization

Adding Elements of an Array



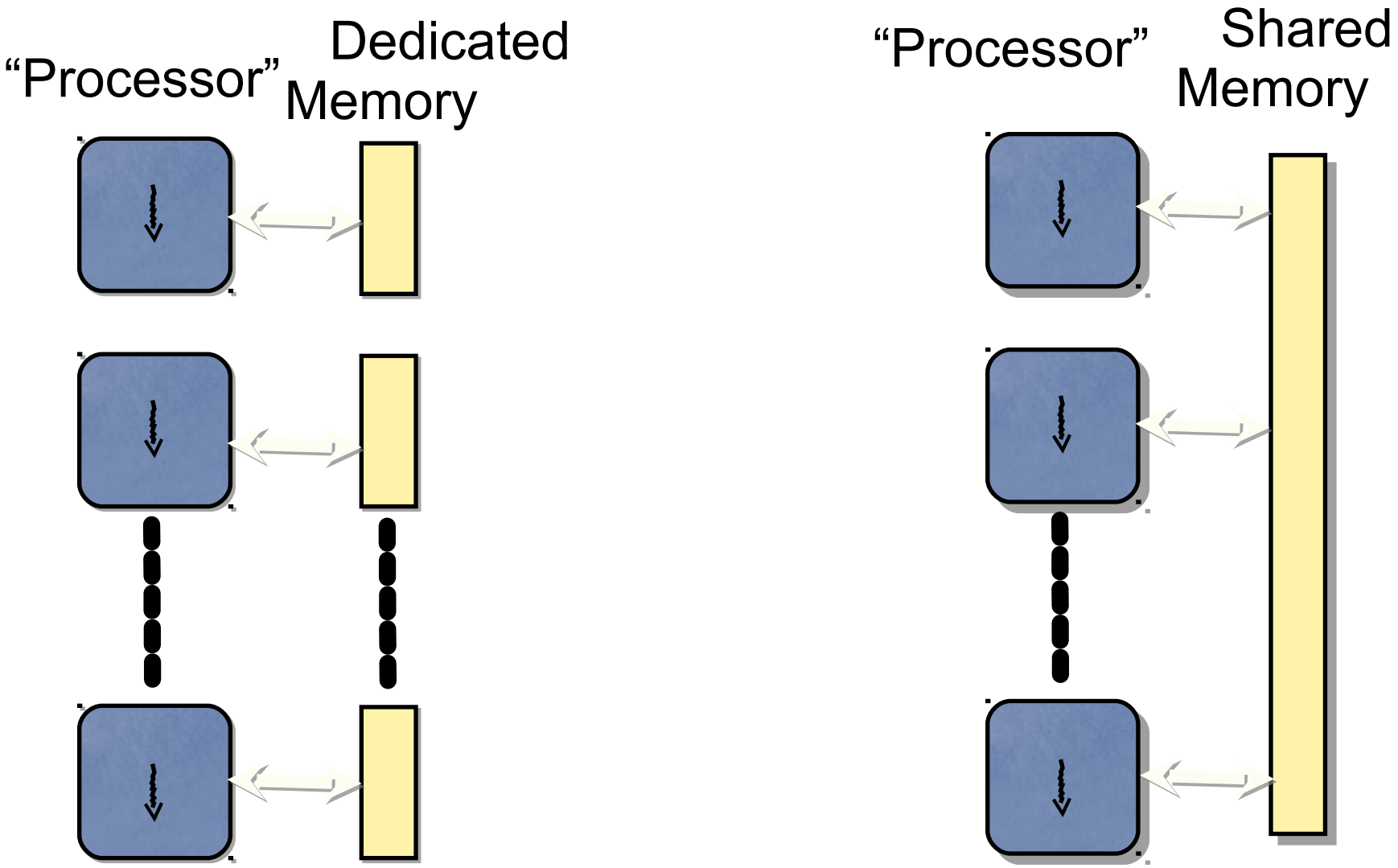
$O(\log_2 n)$

Adding two vectors



Constant Time

Architectural Aspects



- Consider parallelisation on a shared memory machine :

There is a bank balance of Rs. 4,000/-. There are two transanctions : one a cheque payment into the account of Rs. 1,000/- and a credit card expense of Rs. 1,500/-.

Suppose the parallel computer reads the balance first. Then the credit card also reads the balance as Rs. 4,000/-. Add the payment of Rs. 1000/- to it which gives the answer as Rs. 5,000/- Later, the credit card balance gives the remaining balance as Rs. 2,500/- This is wrong as the correct balance is Rs. 3,500/-.

Even if the transactions occur in reverse order then the balance is Rs. 4,500/- which is also wrong.

This is overcome by locking access to the memory to only one processor/process/transaction at a time until it is released by it.

API – the new paradigm

- Application program interface is a software or call that essentially keeps the nitty-gritty details out of the user. It provides an user interface that is simple and details are taken care by the API.

Tour of a normal program execution

- A normal fortran or C program you run needs the following steps :

1) Program writing,

2) Compilation (usually with libraries)

3) Execution

4) Input data/files

5) Output data/files

- Compilation changes the fortran/C code from a high level language into a set of instructions in binary number system (which is what a computer understands).
- It is then loaded into the memory (random access memory, RAM) along with any libraries used from where the instructions are sequentially executed by the CPU.
- Whenever there are data to be read in or written out, then these are transferred from memory to hard disk.

GPGPU

- But now-a-days it is possible to run your program on GPU (graphics processing unit, or your graphics card) instead of CPU(central processing unit).
- Normally GPU calculations are used to render a picture to be put on the screen (which can be very high, higher than the calculations done on CPU) or picture rendering.
- However, when we use it to do our scientific calculations, then it is called GPGPU (instead of the usual special purpose computation – picture drawing/rendering).

What do we require to compute on GPU ?

- Obviously, we need a graphics card or something similar to it.
- We have among others, NVIDIA and ATI.
- We need the software that will help us to talk to GPU.
- A compiler which can do this is nvcc (c compiler made by NVIDIA).
- We need to know this language nvcc.
- Also we need to know how the problem can be parallelized.

Why should we compute on GPU instead of CPU ?

- CPU these days come with multiple cores : dual core, quad core, etc. Communication between the cores in much faster than between CPU is a parallel computer. So, this is better.
- A GPU generally has many more cores than the multicores of the CPU (upto ~500).
- So, you gain enormous speed by using GPU.
- And communication is faster between these cores than between CPUs.
- And it is much, much cheaper.
- So it is talked about as supercomputer on your desktop.
- And as supercomputer for the masses.

- The cores on the GPU are generally somewhat *slower* than the cores on the CPU but they are *so many in number* that they together beat the parallel computer or supercomputer many times over. This has been a general trend towards calculations over past two decades.
- CUDA stands for Compute Unified Device Architecture) helps us to

- The most challenging paradigm-shift required to understand the CUDA programming model is to realize that the old serial model of looping over every element in a data structure does not exist in the CUDA programming style. CUDA kernels replace the body of a loop, and loop constructs with indices are replaced by block and thread size specifications.

```
// A simple example of CUDA to understand threads and blocks

/*     The following code is written in CUDA C, and can be
embedded into a normal C program run on the CPU. It calls a
kernel, which creates two-dimensional blocks of threads of size
BLOCK_SIZE, forming a grid of threads of size GRID_SIZE:

Read more at Suite101:
A Basic NVIDIA CUDA Programming Example
http://www.suite101.com/content/a-basic-nvidia-cuda-programm
ing-example-a236696#ixzz13xKaBOHb
*/

double phi [GRID_SIZE*GRID_SIZE ] ;

dim3 threadsPerBlock (BLOCK_SIZE,BLOCK_SIZE) ;

dim3 numBlocks
(GRID_SIZE/BLOCK_SIZE,GRID_SIZE/BLOCK_SIZE) ;

test<<<numBlocks,threadsPerBlock>>>(phi);
```

Note that the thread blocks could have had any number of dimensions, depending on the specifics of the problem in hand. The kernel function test can then manipulate each element of the array phi[] in parallel:

```
__global__ void test (double *phi ) {

int x = blockIdx.x*BLOCK SIZE + threadIdx. x ;

int y = blockIdx.y *BLOCK SIZE + threadIdx. y ;

phi [ x + BLOCK SIZE*y ] = 3.14159 ;
}
```

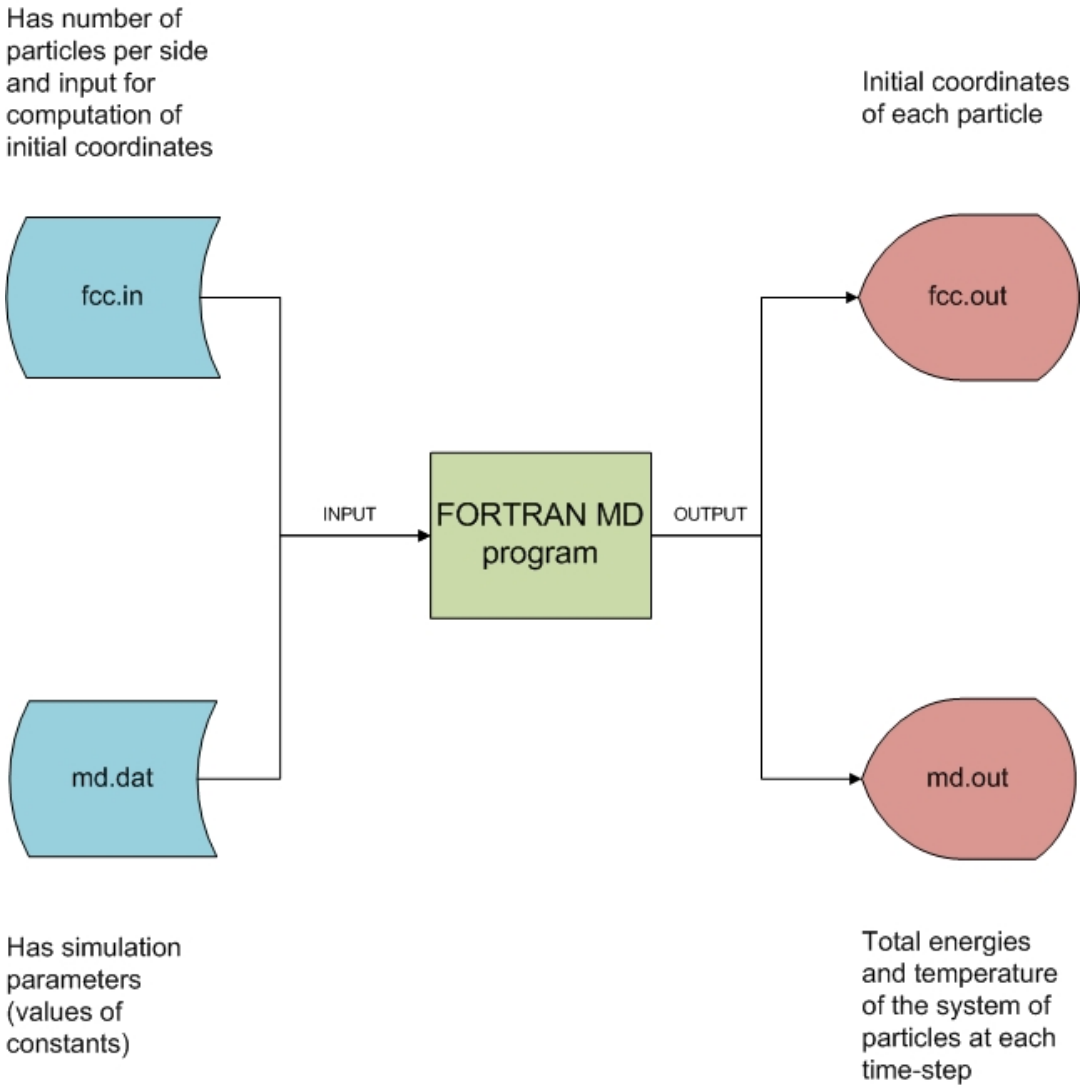
The maximum number of threads in a block and the maximum number of blocks is determined by the particular GPU architecture. All NVIDIA GPUs released to date have a maximum block size of 512 threads. CUDA threads are organized into *warps*. One warp consists of 32 threads, which are executed together. In addition, all the threads in a half-warp of 16 threads can read from memory concurrently in a process known as *coalesced memory access*. Taking advantage of this process is crucial for the optimization of CUDA code.

Example.c and example.cu

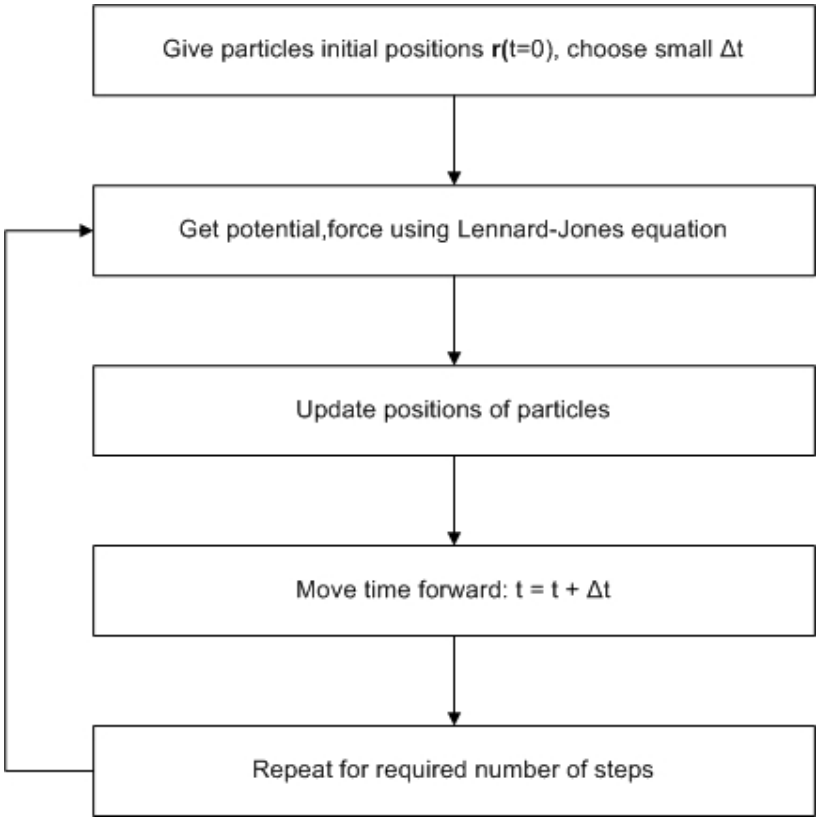
Internship summary

- Goals
  - Parallelize original FORTRAN MD program in CUDA
  - Achieve significant (>100x) speedup
- Steps
  - Convert FORTRAN program to C
  - Parallelize the C program

FORTRAN program



FORTRAN MD program algorithm

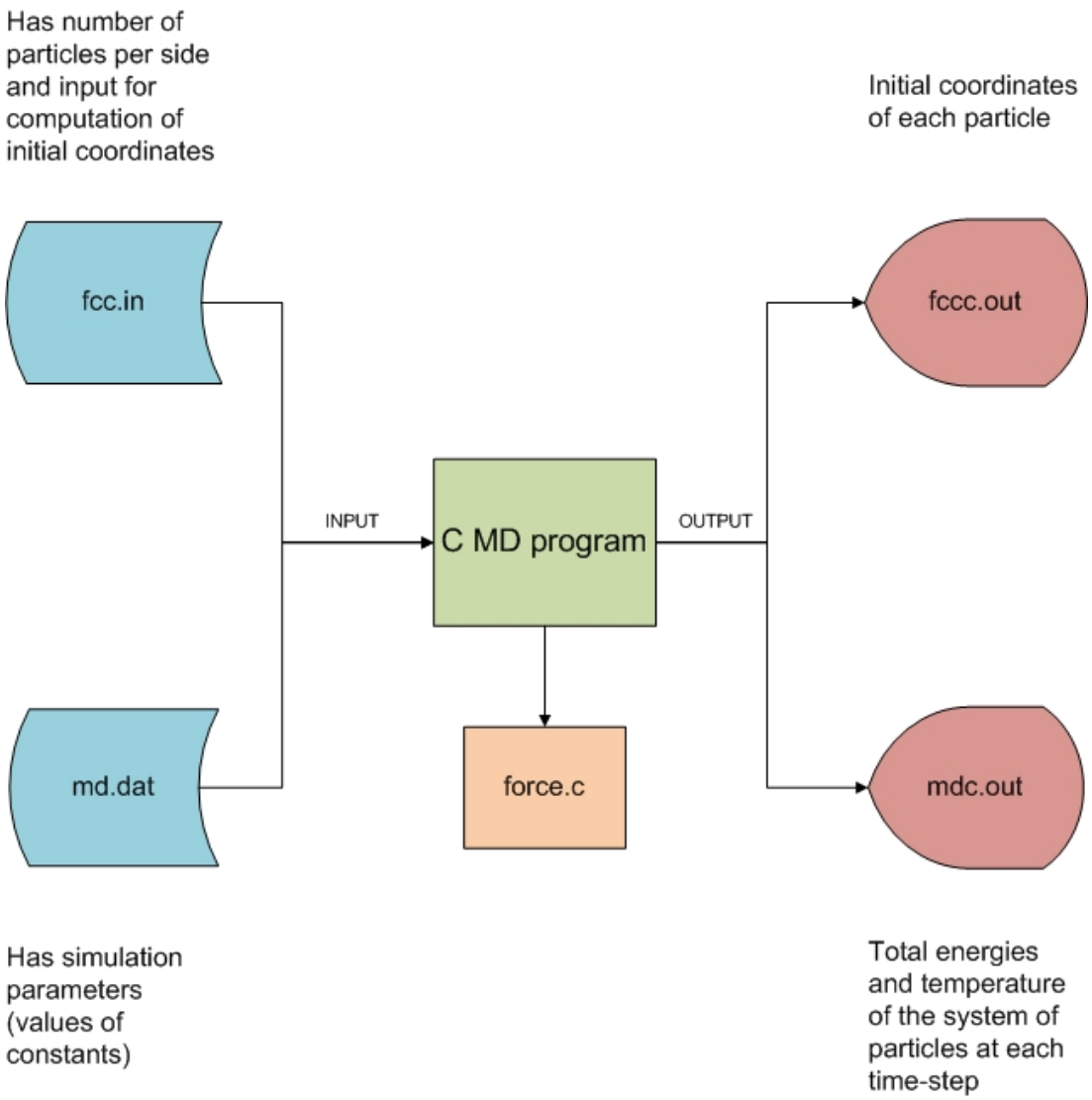


Converting FORTRAN to C

- Structure of FORTRAN program largely preserved, but functions split and stored in separate files
- Module for common variables converted to a C **structure**
- C st

```
struct comn_vbles
{
    long i, j, n, ok;
    double epskj, epsamu, epsamu_24, epskj_4, pot, sigma, s2, s6, s12, l;
    double *rx, *ry, *rz, *vx, *vy, *vz, *fx, *fy, *fz;
};
```

C program


Has number of particles per side and input for computation of initial coordinates
Initial coordinates of each particle
fcc.in
fccc.out
INPUT
C MD program
OUTPUT
md.dat
force.c
mdc.out
Has simulation parameters (values of constants)
Total energies and temperature of the system of particles at each time-step

Results

- Results of C & FORTRAN programs compared
- Fluctuations in all energy computations in C program were less than FORTRAN
- Added time estimation and progress indication features in C program
- C program runs slightly faster than FORTRAN
- Discussion of results in accompanying PDF

Parallelizing *force* function in C program

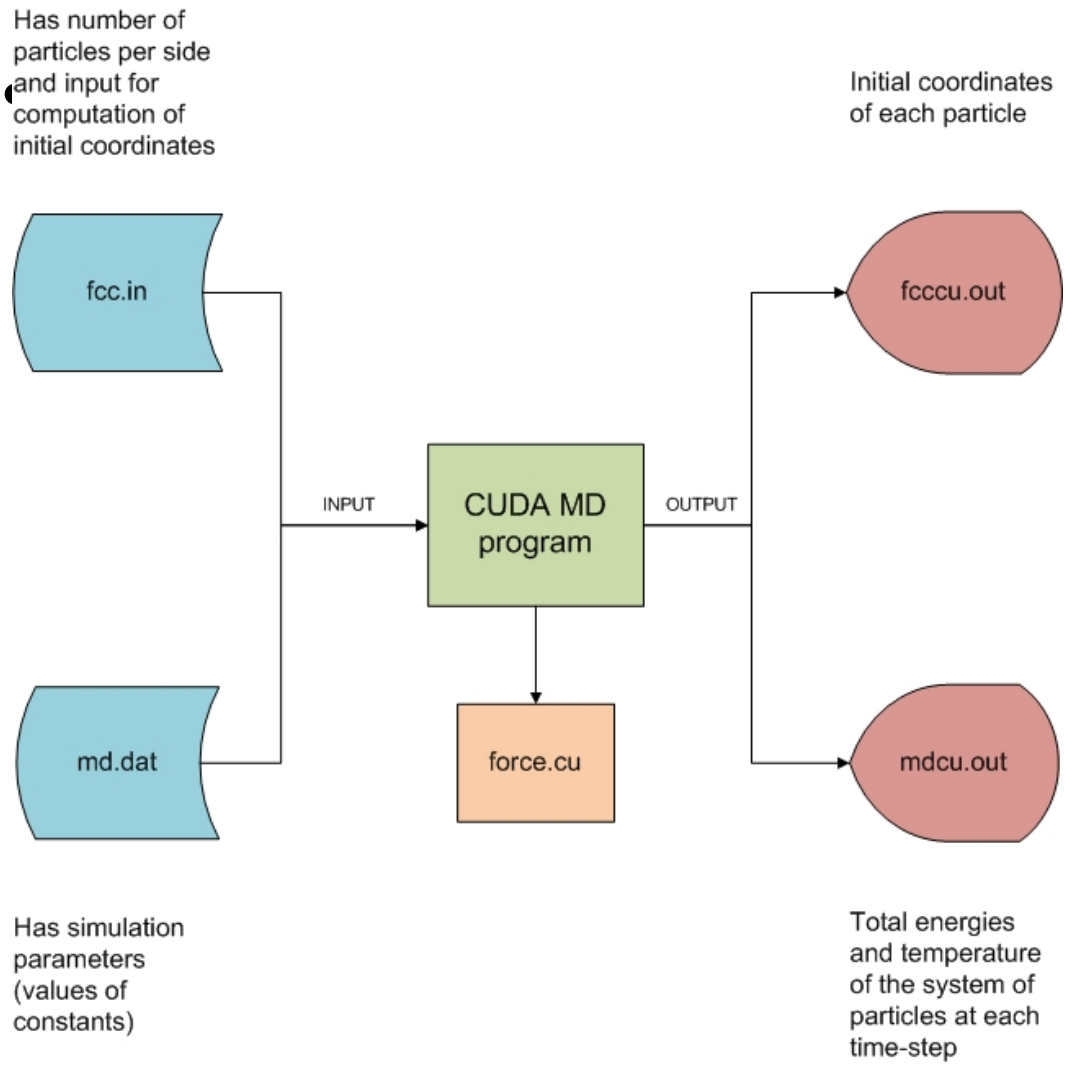
- Computed force acting on each particle pair in parallel
  - Copied force and position vectors to device memory
  - Computed force acting on each particle on separate threads
  - Total potential energy was computed as a sum of individual potential energies of each particle
    - All particles contribute to total potential energy which was stored in a single variable
    - To avoid race condition, device computes potential energy vectors and host function computes the sum of the vector elements

Device memory variables

```
struct device_scalars
{
    double pot, sigma, s2, s6, s12, l;
};

struct device_vectors
{
    double *rx, *ry, *rz, *vx, *vy, *vz, *fx, *fy, *fz, *pot;
};
```

CUDA program



Blocks and threads

- Four approaches used:
  - 1. One thread, many blocks: Low speedup over C program (5x), because blocks sequentially schedule
  - 2. Many blocks, fewer threads: Higher speedup, but low threads per block
  - 3. Many threads, fewer blocks: Higher speedup, but many cores are idle
  - 4. Maximum blocks and threads: Best speedup results
    - Execution times for 13500 particles, 3000 steps
      - 1. 13500 blocks, 1 thread: 1 hour 15 minutes
      - 2. 375 blocks, 36 threads: 3 minutes 40 seconds
      - 3. 27 blocks, 500 threads: 2 minutes 20 seconds
      - 4. 448 blocks, 512 threads: 2 minutes 15 seconds

Results

- Fluctuation in energies and temperature similar to C program
- Since outer *for* loop was parallelized, computation reduced from $O(n^2)$ to $O(n)$
- Achieved peak 198x speedup over C program
- Detailed results in accompanying PDF file

MDTimes and Mdgraphs