Parallel Programming

Salvatore Filippone

salvatore.filippone@cranfield.ac.uk

Parallel Programming

How do you build a program to work in parallel?

We mainly have two options:

1. Multiple (cooperating?) processess (i.e. memory is *logically* distributed);
2. Multiple (cooperating?) threads (i.e. memory is *logically* shared)

Variations:

- Memory is private, data must be sent explicitly (MPI);
- Memory is by default private, but can be logically shared (PGAS);
- Memory is by default shared, but can be private (OpenMP).

Processes

Process

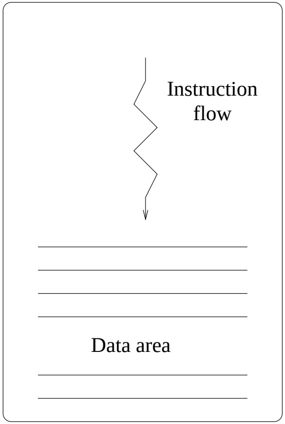
a dynamic entity: a program in execution (i.e. a flux of instructions) with its memory resources.

Memory

a data area (visible to the programmer), plus a reserved area for the state of the process (not directly visible).

Major issues

Data access and synchronization



Processes

Multiprogramming

Existence of several independent processes in a same computing unit; their execution appears (macroscopically) to be simultaneous:

1. Interaction between applications and operating system (scheduling)
2. Use cases, application mix;

In typical CSE simulations (a portion of the) machine is dedicated, only the minimal set of operating system utilities runs alongside the user program.

Threads

Threads

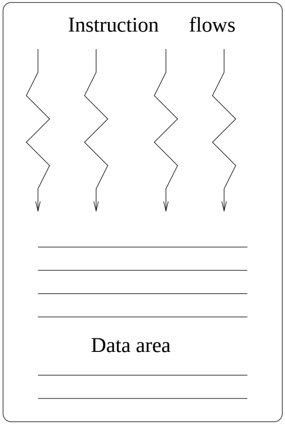
Multiple instruction fluxes within a given process.

Memory

common data plus private thread data (at minimum: processor state)

Major issues

Data access and synchronization



Where is the data?

Distributed Shared Memory (DSM)

- Some distributed memory multicomputer systems have a single address space in which the available addresses are partitioned across the memory banks.
- Others have multiple address spaces in which each CPU is able to issue addresses only to its 'own' local memory bank.
- The operation of the single address space version of this architecture, known as distributed shared memory (DSM), is logically unchanged from the shared memory multiprocessor schemes.

Where is the data?

non-uniform memory access (NUMA)

- Note that some memory accesses only need to go to the physically attached local memory bank, while others, according to the address, have to go through the interconnect. This leads to different access times for different memory locations, even in the absence of contention.
- This latter property makes distributed shared memory a non-uniform memory access (NUMA) architecture. For high performance, it is essential to place code and data for each thread or process in readily accessible memory banks.

Where is the data?

Message Passing

In multiple address space versions (known as distributed memory or DM) co-operative parallel action has to be implemented by the software (at least at the level of the run-time system). This is called *message passing*, as it is based on *send* and *receive* primitives for exchanging data (*messages*)

Programming models

The programming model is *more immediately relevant* for the programmer than the physical architecture.
Most relevant types:

Implicit Parallelism: Leave everything to the compiler;

Data parallel: A single flux of control, arithmetic operations applied to data sets (e.g. HPF) with a shared address space;

Message passing: A set of processes that coordinate themselves by exchanging *messages* containing data;

Shared variable: Shared address space, multiple control fluxes.

PGAS Partitioned Global Address Space languages are somewhat in between: a set of processes (private data) that coordinate themselves through a set of (selectively) shared data objects.

Parallel programming tools

How do we produce the code?

1. Support libraries;
2. Compilers (and language extensions).

Examples:

MPI:	Message Passing Interface: a standardized support library.
Pthreads:	Posix Threads; a standardized support library for shared memory programming
OpenMP, OpenACC,CUDA:	Shared memory programming extensions via compiler directives;
UPC, CAF:	Universal Parallel C, CoArray Fortran: PGAS languages;

From Serial to Parallel

How do you parallelize a code?

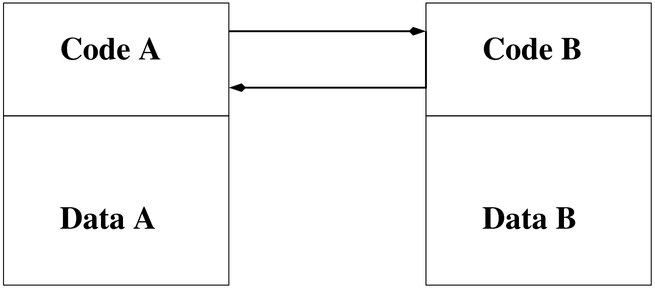
Linking sequential programs together using *message-passing*.
: Because the executable image of a sequential program is called a process, this parallel scheme is termed *process-based parallel programming*. The difficulty with this model of execution is the high cost of establishing the underlying parallel processes.

Splitting the memory associated with a sequential program into multiple parts, one part of which is shared by the others.
: The part that is split is the data part of the sequential executable image, so this parallel scheme is often called *data-sharing*. The scheme is also termed *thread-based* parallel programming. The difficulty lies in avoiding race conditions and getting performance right.

Actually, *splitting the data* is also a critical issue in message-passing.

Two processes

- Launch two different (but related) processes;
- Have them exchange data by *messages*: when process A needs some data, process B sends it (and viceversa);



Sending messages

In order to send a message from Process A to Process B, the following actions must occur:

- Process A executes a SEND command specifying:
- the type of message it wants to send;
- the location of the start of the message in the Data part of Process A's memory; and
- the destination process to which the message is to be sent (i.e. Process B).

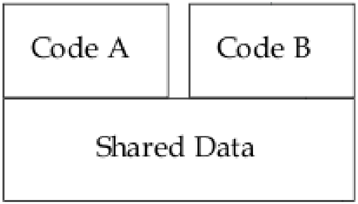
The runtime system associated with Process A accesses the message from Process A's memory and sends it to Process B when required, optionally putting the message in a buffer before it gets sent. Process A cannot execute further code until the message has been completely read from its Data memory (but does not necessarily wait for its delivery to system B).

Receiving a Message

- Process B executes a RECEIVE command specifying:
    - the type of message it wants to receive;
    - the location of the start of the place in the Data part of Process B's memory in which it wishes the message to be put; and
    - the source process from which the message is to be received (optional - the Process may be happy to receive from any other Process).
- The runtime system associated with Process B synchronises with the runtime system associated with Process A and transfers the message to the required location in Process B's Data memory. Process B cannot execute further code until the entire message has been written into its Data memory.
- If Process B executes its RECEIVE command before Process A executes its SEND command, Process B will temporarily halt until the message becomes available. If there is no buffer in the runtime system, the reading of the sent message does not start until the receiving process executes its RECEIVE command.

Two-fold Parallelism

Within a single process, an obvious way of allowing two-fold parallel execution is to allow two program counters to control progress through two separate, but related, code states. To a first approximation, the two streams of instructions will need to share the sequential data state.



When, as frequently happens, Code A and Code B are identical, this scheme is termed single-program, multiple-data (SPMD).

Physical vs. logical memory sharing

Physical memory	**Logical Memory**	**Example**
non shared	non shared	*Message Passing*
non shared	shared	*Distributed Shared Memory* *PGAS* Languages
shared	non shared	*Message Passing*
shared	shared	P-threads Directives (OpenMP & friends)

Which tool to use?

It is commonly assumed and stated that:

Shared memory programming is easier

I personally prefer to state another principle:

Conservation of effort

Given the same serial program, parallelization via OpenMP and MPI takes the same amount of effort, but the distribution of work in the two phases of:

1. Initial parallel version;
2. Performance tuning;

is specular, i.e. with MPI the vast majority of the effort is spent in the initial parallelization, whereas with OpenMP the reverse is often true.

This is yet another instance of the Pareto principle 80-20; it is not an absolute truth, but close to actual experience.