Overview of parallel programming paradigms

Albert-Jan Yzelman

29th of November, 2013

KATHOLIEKE UNIVERSITEIT LEUVEN

Flynn's Taxonomy

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Examples:

- classical von Neumann computing; program and data streamed to processor.
- vectorisation (MMX/AVX/...), vector computers.
- shared-memory programming, i.e.,
  - ccNUMA architectures,
  - symmetric multiprocessors,
  - non-cc NUMA (scratchpad memory, Cell Broadband Engine, ...)
- distributed-memory programming (classical, grids, clouds, ...).
- hybrid parallel programming.

Programming models

A class on parallel computing focuses on MIMD.

- Two flavours of MIMD:
  - Single Program, Multiple Data (SPMD), and
  - Multiple Program, Multiple Data (MPMD).
- **Both** SPMD and MPMD programs can be in **lockstep**.

Examples:

- GPUs often use **stream-based** programming. This is SPMD and employs groupwise lockstepping.
- Past and future architectures do allow and even **promote MPMD programming**: e.g., the Cell BE and the Epiphany archtictures.
- BSP is an SPMD paradigm.

Common design patterns

Often-used techniques for writing parallel software:

- task-based: identify independently executable work and group them into tasks, then schedule these dynamically. This may include modeling inter-task dependencies in terms of input- and output-data. In the extreme: workflows.
- divide-and-conquer: concurrently conquer a divided problem.
- geometric decomposition: often used in spatial computational science simulations.
- pipelining: like task-based parallelism, but with serial-dependencies; this pattern sometimes appears in a subsection of a parallel program.
- master/worker: one process assignes work others.
- fork/join: after a fork a new process appears, starting at the same part of the program.

Task-based parallelisation

A task has three states: (1) wait for data, (2) ready, and (3) done. If $\mathcal{T}_i$ is the set of tasks in state i, then a simple scheduler runs a loop similar to:

- While $\mathcal{T}_2$ is empty, wait.
- Remove a task t from $\mathcal{T}_2$.
- Execute t.
- For all tasks $s \in \mathcal{T}_1$, if s waits on t,
- let s know t is done, and
- if s has no further waits, remove it from $\mathcal{T}_1$ and add it to $\mathcal{T}_2$.
- Insert t into $\mathcal{T}_3$.

This requires thread-safe datastructures (concurrent removals, insertions). **The orange statement is critical to performance**.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.

Remark: the sequential running time $T_{\text{seq}} = \sum_{v \in V} w(v)$.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.

Remark: let $P \subset V$ be a path from r to a **leaf vertex** x s.t.

$$\forall v \in V, \quad \nexists (x, v) \in E,$$

then $w(P) = \sum_{v \in P} w(v)$ is the **path weight**. The weighted **critical path** P_c has for all other possible paths P that $w(P_c) \geq w(P)$.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.
- find the weight of the weighted critical path (and try to minimise this for scalability).

Remark: if $p \to \infty$, the minimum amount of computation remains $w(P_c)$, **always**.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.
- find the weight of the weighted critical path (and try to minimise this for scalability).

Remark: if $p \to \infty$, the minimum amount of computation remains $w(P_c)$, **always**. The parallel compute time T_p is bounded by

$$T_p = \mathcal{O}(w(P_c)/r + T_{\text{seq}}/p),$$

with r the computation speed of a single worker.

Common HW/OS capabilities

Hardware and operating systems often support parallelism by:

- thread and process execution and control:
  - spawn,
  - abort,
  - wait for,
  - exit.
- mutexes (locks),
- synchronisation barriers,
- transactional memory
  - readily available: atomics,
  - more complex technologies: speculative threading.

In-software equivalents of the above are possible (for instance, the use of spin-locks instead of barriers).

Communication paradigms

Popular ways for inter-process communication:

- message passing (MPI, BSP),
- direct remote memory access (BSP, MPI),
- collectives (MPI),
- global arrays / global address space (GA/PGAS/UPC).

These approaches can be

- Blocking or non-blocking,
- One-sided or two-sided (and up to p-sided).

Classically,

- MPI is a two-sided blocking message passing (send/receive pairs) paradigm, while
- BSP is a one-sided non-blocking direct remote memory access paradigm.

Naturally, both paradigms have evolved beyond this strict classification.

History of MPI

- 1994: Message Passing Interface (MPI) became available as a standard interface for parallel programming in C and Fortran 77.
- Designed by a committee called the MPI Forum consisting of computer vendors, users, computer scientists.
- Based on sending and receiving messages by a pair of processors. One processor sends; the other receives. Both are active in the communication.
- Underlying model: communicating sequential processes (CSP) proposed by Hoare in 1978.
- MPI itself is not a model. BSP is a model.
- MPI is an interface for a communication library, like BSPlib.

Recent history of MPI

- 1997: MPI-2 standard defined. Added functionality:
  - one-sided communications (put, get, sum)
  - dynamic process management
  - parallel input/output
  - languages C++ and Fortran 90
- 2003: first full implementations of MPI arrive, namely MPICH (Argonne National Labs) and LAM/MPI (Indiana University).
- 2004–: Open MPI. Open-source project, merges 3 MPI implementations: LAM/MPI, FT-MPI (University of Tennessee), LA-MPI (Los Alamos National Laboratory).
- Many users still use MPI-1, particularly its latest version MPI-1.2.

Why use MPI?

- It is available on almost every parallel computer, often in an optimised version provided by the vendor. Thus MPI is the most portable communication library.
- Many libraries are available written in MPI, such as the numerical linear algebra library ScaLAPACK.
- You can program in many different ways using MPI, since it is highly flexible.

Why not?

- It is huge: the full standard has about 300 primitives. The user has to make many choices.
- It is not so easy to learn. Usually one starts with a small subset of MPI. Full knowledge of the standard is hard to attain.
- MPI-2 has not widely been accepted (yet), nor has it been fully implemented in every MPI library. If you like one-sided communications you may want to consider BSPlib as an alternative.

Ping pong benchmark

- The cost of communicating a message of length n is

$$T(n) = t_{\text{startup}} + nt_{\text{word}}.$$

  Here, t_{startup} is a fixed startup cost and t_{word} is the additional cost per data word communicated.
- Communication of a message (in its blocking form) synchronises the sender and receiver. This is pairwise synchronisation, not global.
- Parameters t_{startup} and t_{word} are usually measured by sending a message from one processor to another and back: ping pong.
- The message length is varied in the ping pong benchmark.
- There is only one ping pong ball on the table.

Send and receive primitives

```
if (s==2)
  MPI_Send(x,5,MPI_DOUBLE,3,0,MPI_COMM_WORLD);
if (s==3)
  MPI_Recv(y,5,MPI_DOUBLE,2,0,MPI_COMM_WORLD,
           &status);
```

- Processor $P(2)$ sends 5 doubles to $P(3)$.
- $P(2)$ reads the data from its array `x`. After transmission, $P(3)$ writes these data into its array `y`.
- The integer '0' is a tag for distinguishing between different messages from the same source processor to the same destination processor.
- `MPI_Send` and `MPI_Recv` are of fundamental importance in MPI.

Communicator: the whole processor world

```
if (s==2)
  MPI_Send(x,5,MPI_DOUBLE,3,0,MPI_COMM_WORLD);
if (s==3)
  MPI_Recv(y,5,MPI_DOUBLE,2,0,MPI_COMM_WORLD,
           &status);
```

- A communicator is a subset of processors forming a communication environment with its own processor numbering.
- `MPI_COMM_WORLD` is the communicator consisting of all the processors.

Send/Receive considered harmful

- 1968: Edsger Dijkstra, guru of structured programming, considered the Go To statement harmful in sequential programming.
- Go To was widely used in Fortran programming in those days. It caused spaghetti code: if you pull something here, something unexpected moves there.
- No one dares to use Go To statements any more.
- Send/Receive in parallel programming has the same dangers, and even more, since several diners eat from the same plate.
- Pull here, pull there, nothing moves: deadlock.
- Deadlock may occur if $P(0)$ wants to send a message to $P(1)$, and $P(1)$ to $P(0)$, and both processors want to send before they receive.

Inner product program mpiinprod

```
int main(int argc, char **argv){

    int p, s, n;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&s);

    if (s==0){
        printf("Please enter n:\n");
        scanf("%d",&n);
        if(n<0)
            MPI_Abort(MPI_COMM_WORLD,-1);
    }
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

Collective communication: broadcast

```
MPI_Bcast(&n, 1,    MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(buf,count,datatype,root,communicator);
```

- Broadcast `count` data items of a certain `datatype` from processor `root` to all others in the `communicator`, reading from location `buf` and also writing it there.
- All processors of the communicator participate.
- Extensive set of collective communications available in MPI. Using these reduces the size of program texts, and allows hardware vendors to provide optimised algorithms.

Inner product program mpiinprod (cont'd)

```
...
nl= nloc(p,s,n);       //local vector length
x = vecallocd(nl);     //allocate and initialise x
for (i=0; i<nl; i++) {
    iglob= i*p+s;
    x[i]= iglob+1;
}

MPI_Barrier(MPI_COMM_WORLD); // global sync for timing
time0=MPI_Wtime();           // wall clock time
...
```

Inner product program mpiinprod (cont'd)

```
...

//do calculate inner product
alpha= mpiip(p,s,n,x,x);

//sync for timing
MPI_Barrier(MPI_COMM_WORLD);
time1=MPI_Wtime();

...
MPI_Finalize();
exit(0);
```

Inner product function mpiip

```
double mpiip(int p,int s,int n,
             double *x,double *y){

    double inprod, alpha;
    int i;

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++)
        inprod += x[i]*y[i];
    MPI_Allreduce(&inprod,&alpha,1,MPI_DOUBLE,
                  MPI_SUM,MPI_COMM_WORLD);

    return alpha;
}
```

Collective communication: reduce

```
MPI_Allreduce(&inprod, &alpha,  1,  MPI_DOUBLE,
              MPI_SUM,   MPI_COMM_WORLD);

MPI_Allreduce(sendbuf, recvbuf, count, datatype,
              operation, communicator);
```

- The reduction operation by `MPI_Allreduce` sums the double-precision local inner products `inprod`, leaving the result `alpha` on all processors.
- One can also do this for an array instead of a scalar, by changing the parameter 1 to the array size `count`, or perform other operations, such as taking the maximum, by changing `MPI_SUM` to `MPI_MAX`.

Collectives

More collective operations:

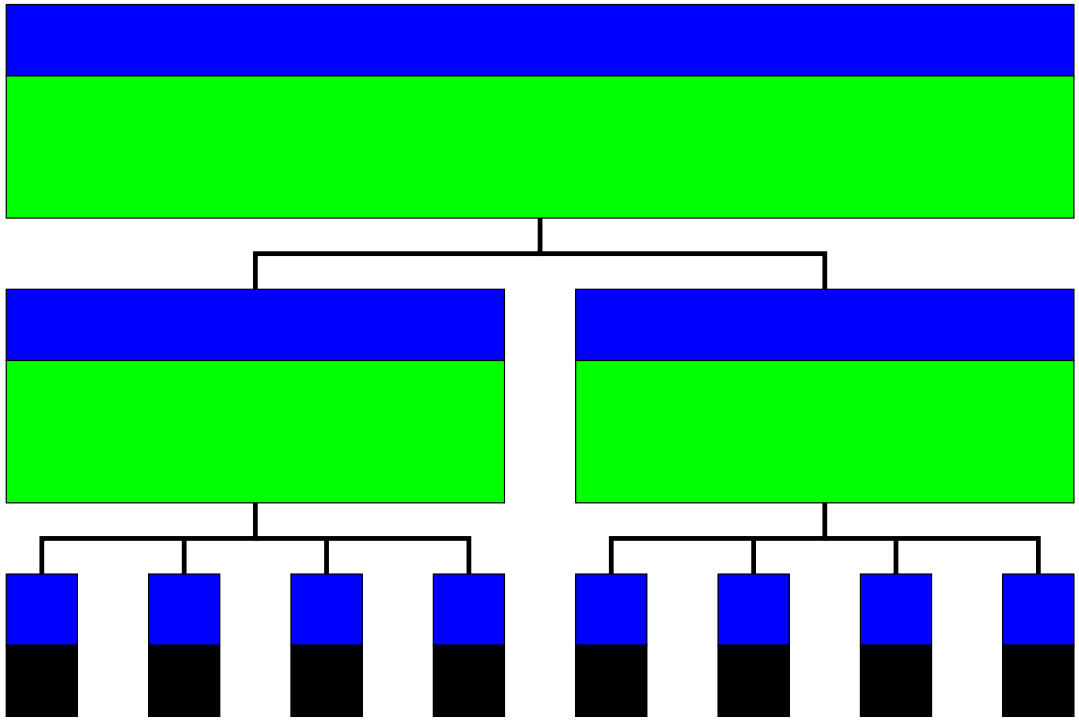
- broadcast,
- scatter,
- gather,
- reduction (also reduce-scatter),
- all-gather, and
- all-to-all.

For a nice graphical overview, see
`www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node64.html`

- MPI directly integrates these patterns by providing primitives: MPI_BCAST, MPI_GATHER, MPI_SCATTER, ...
- MPI also supports collectives on vectors instead of single entities (MPI_GATHERV. MPI_SCATTERV, ...), thus enabling optimisations such as the two-phase broadcast.

Multi-BSP

Multi-BSP: A BSP computer consists out of p other BSP computers (Valiant, 2008), **or** p processors.



(Blue: local memory, green: network interconnect, black: processors)

- MulticoreBSP for C already supports nested BSP runs.

Multi-BSP

The BSP FFT splits local computation into two local (generalised) FFTs, with one data redistribution phase in between.

FFT example: instead of reverting to optimised sequential FFTs, we revert to **parallel** FFT kernels.

Consider an 8 socket machine with eight quadcore processors.

BSP FFT: two comp. phases, one comm. phase.
Multi-BSP FFT: three comp. phases, two comm. phases:

- start 8 BSP FFT processes, which each recursively call

Multi-BSP

The BSP FFT splits local computation into two local (generalised) FFTs, with one data redistribution phase in between.

FFT example: instead of reverting to optimised sequential FFTs, we revert to **parallel** FFT kernels.

Consider an 8 socket machine with eight quadcore processors.

BSP FFT: two comp. phases, one comm. phase.
Multi-BSP FFT: three comp. phases, two comm. phases:

- start 8 BSP FFT processes, which each recursively call
- a parallel BSP FFT using 4 cores, which each recursively call

Multi-BSP

The BSP FFT splits local computation into two local (generalised) FFTs, with one data redistribution phase in between.

FFT example: instead of reverting to optimised sequential FFTs, we revert to **parallel** FFT kernels.

Consider an 8 socket machine with eight quadcore processors.

BSP FFT: two comp. phases, one comm. phase.
Multi-BSP FFT: three comp. phases, two comm. phases:

- start 8 BSP FFT processes, which each recursively call
- a parallel BSP FFT using 4 cores, which each recursively call
- an optimised sequential FFT.

Multi-BSP

The BSP FFT splits local computation into two local (generalised) FFTs, with one data redistribution phase in between.

FFT example: instead of reverting to optimised sequential FFTs, we revert to **parallel** FFT kernels.

Consider an 8 socket machine with eight quadcore processors.

BSP FFT: two comp. phases, one comm. phase.
Multi-BSP FFT: three comp. phases, two comm. phases:

- start 8 BSP FFT processes, which each recursively call
- a parallel BSP FFT using 4 cores, which each recursively call
- an optimised sequential FFT.

While this introduces **more** data redistribution stages, the BSP g and l are lower in each of these step; each redistribution step is **cheaper**.

Summary

We have seen

- various ways on exploiting parallelism on current hardware,
- a high-level overview of parallel programming paradigms,
- common design patterns used within these paradigms,
- what the hardware and OS can do for you,
- an overview of communication paradigms,
- how to analyse fine-grained parallel applications,
- a short introduction to the Message Passing Interface, and programming using MPI, and
- what the future of BSP might look like.