Parallelism paradigms

Intro part of course in Parallel Image Analysis

Elias Rudberg

elias.rudberg@it.uu.se

March 23, 2011

Outline

1 Parallelization strategies

2 Shared memory

3 Distributed memory

4 Parallelization using GPUs

5 Speedup and scaling

Parallel computing

Traditionally, a computer program has been considered as something serial: one instruction executed after the other in a sequence.

However, during last $\sim$ 20 years parallel computing has become increasingly important.

```
ISI Web of Science search for "Parallel computing" in topic.
1961-1965   Results:    1
1966-1970   Results:    0
1971-1975   Results:    1
1976-1980   Results:    6
1981-1985   Results:   14
1986-1990   Results:   66
1991-1995   Results:  700
1996-2000   Results: 1326
2001-2005   Results: 1708
2006-2010   Results: 2316
```

"What is the best way to parallelize?"

No good answer to the question "What is the best way to parallelize?" since this depends on so many things.

Difficult to say anything general since:

- Many different hardware architectures exist, having different properties regarding computing performance, communication overhead, memory availability, disk space, etc.
- Algorithms are so different; a parallelization approach that works for one algorithm may be useless for another algorithm.
- Interplay between chosen algorithm and hardware properties also important.

"Best" algorithm may depend on hardware.

When considering only *serial* programs, there is often one algorithm that is "the best" algorithm to use for solving a given problem, in the sense that the problem is then solved using as few operations as possible.

However, when *parallel* implementations are considered, it may turn out that the algorithm that uses the smallest number of operations may not be the fastest algorithm for solving the problem. If the "best" algorithm is difficult to parallelize, the problem may be solved faster using an algorithm that performs more work, but is easier to parallelize.

Key: find independent chunks of work

All parallelization strategies somehow make use of the fact that the work can be divided into (more or less) independent parts. If the algorithm does not allow that, then consider using another algorithm.

Common approaches

The perhaps most common parallelization approach is to use a predefined static division of the work. Example: finite-difference computations.

Another common choice is to use a task-based approach, where chunks of work are defined as "tasks" and are somehow scheduled to run on different cores/nodes.

Master/slave model

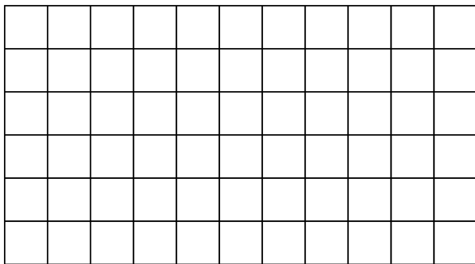
[Draw this on blackboard.]

Good: tasks can be general. Load balancing issues can be solved.

Drawback: master becomes bottleneck when using a large number of workers.

Fixed division of work

Example: grid divided into boxes.



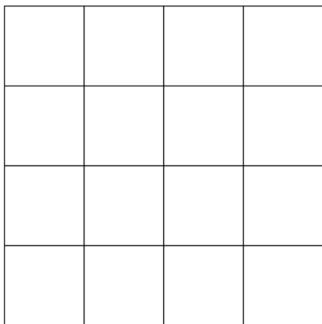
Good: no “master” overhead – can work well for large number of workers.

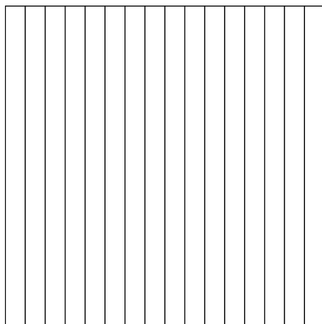
Drawback: load balancing problems if not same amount of work in each cell.

Fixed division of work

Splitting work in 1D or 2D

4x4 vs 16x1:





In this case, 2d splitting gives larger independent chunks of work.

Choosing algorithm depending on hardware

Purification vs diagonalization

Example of problem where parallelization issues motivate using a different algorithm: density matrix construction in electronic structure calculations.

Two algorithms:

- Diagonalization. Small amount of work, but difficult to parallelize.
- Purification, based on matrix multiplication. Larger amount of work, but also easier to parallelize.

Purification $\sim$ 3-4 times slower in serial program, but becomes preferable when parallelization is considered.

Fault tolerance

An important advantage of task-based approaches is that if something goes wrong, you can fix it by just re-running the failed task, without need to restart the entire calculation.

This becomes more and more important for larger clusters.

Shared vs distributed memory

Two main types of parallelization:

- *Shared memory* parallelization
- *Distributed memory* parallelization

Two main types of parallelization

Shared memory parallelization (for multicore computers):

- Each core has access to the same memory as all the other cores.
- The program uses threads to exploit the cores, using POSIX threads (pthreads) or OpenMP.

Distributed memory parallelization (for compute clusters):

- The program consists of several processes running on separate nodes.
- Each process can only access its own local memory; communication needed to interchange information between processes. Message Passing Interface (MPI)

Outline

1 Parallelization strategies

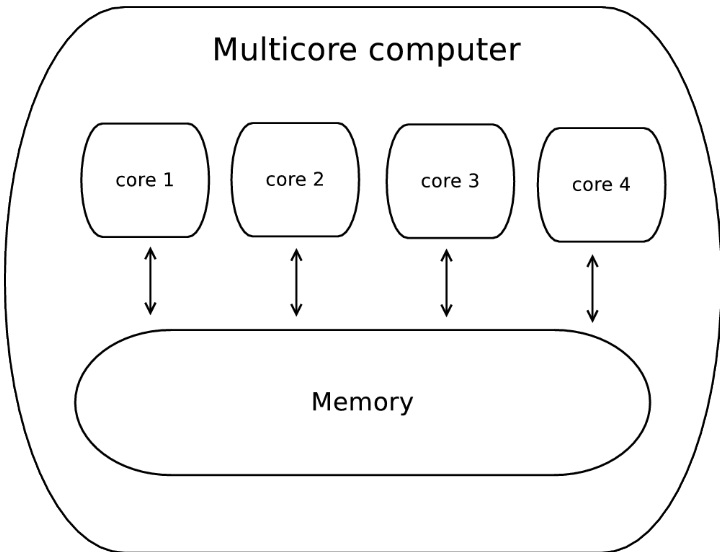
2 Shared memory

3 Distributed memory

4 Parallelization using GPUs

5 Speedup and scaling

Shared memory (I)



Shared memory (II)

Ways to implement shared-memory parallelization:

- POSIX threads (pthreads)
- OpenMP

(There is also the possibility of using the fork() function, but we will focus on pthreads and OpenMP.)

Shared memory

Communication between threads (I)

Simplest case: each thread is only writing to memory locations that are not accessed by other threads. Then, all synchronization that is needed is to somehow determine when all threads have completed their work, and then (serially) gather the results.

[Blackboard: illustrate independent threads.]

Shared memory

Communication between threads (II)

Trickier case: all threads produce results that are to be stored in a common location. Then programmer must make sure that there is no conflict when different threads try to write to the same result buffer.

[Blackboard: illustrate threads synchronizing to write to common buffer.]

Possible solution: let each thread store results in a separate buffer, join results serially in the end.

Shared memory

POSIX threads (pthreads)

Example of thread creation using the pthread (POSIX) standard:

```
/* start threads */

for(int i = 0; i < noOfThreads; i++) {
    pthread_create(&threadParamsList[i]->thread, NULL,
                   execute_joblist_J_std_thread_func,
                   threadParamsList[i]);
} /* END FOR i */

do_output("threads started OK.");

/* wait for threads to finish */

for(int i = 0; i < noOfThreads; i++) {
  pthread_join(threadParamsList[i]->thread, NULL);
} /* END FOR i */

do_output("all threads have finished.");
```

Shared memory

OpenMP

Example of thread creation using OpenMP:

```
static void do_naive_mmul(std::vector<ergo_real> & C,
                          const std::vector<ergo_real> & A,
                          const std::vector<ergo_real> & B,
                          int n) {
#pragma omp parallel for default(shared)
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++) {
      ergo_real sum = 0;
      for(int k = 0; k < n; k++)
        sum += A[i*n+k] * B[k*n+j];
      C[j*n+i] = sum;
    }
}
```

Shared memory

Pthreads/OpenMP confusion

Sometimes the term "OpenMP" is used when what is really meant is "threading" (pthreads or OpenMP).

For example, sometimes people talk about "MPI or OpenMP" when discussing distributed memory vs shared memory parallelization.

Shared memory

Running jobs

A threaded program should be submitted to the queueing system as a "node" job, by using the following line in the job script:

```
#SBATCH -p node
```

Alternatively, you may specify `-p devel` or run your threaded program after getting a dedicated node using the `interactive` command.

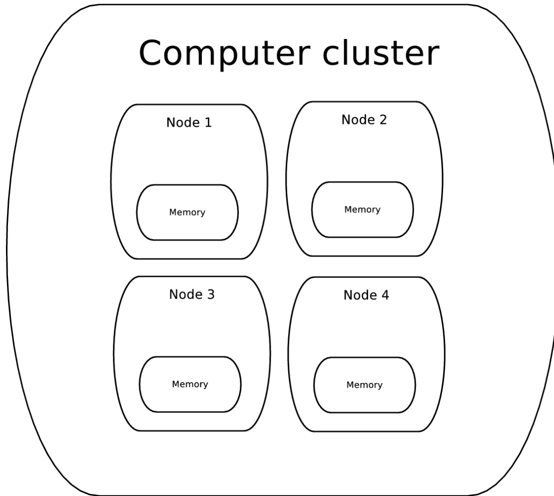
Note: you should *not* use `-p core` for a threaded program, since a threaded program violates the rules for `-p core` jobs.

UPPSALA UNIVERSITET

Outline

1. Parallelization strategies
2. Shared memory
3. Distributed memory
4. Parallelization using GPUs
5. Speedup and scaling

Distributed memory (I)

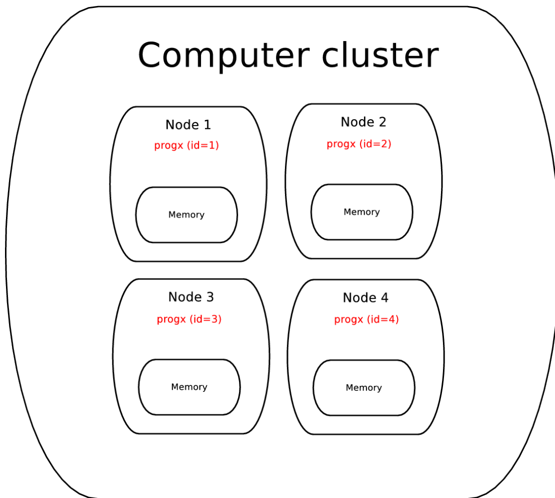


Distributed memory (II)

A program parallelized for distributed memory runs as a separate process on each compute node.

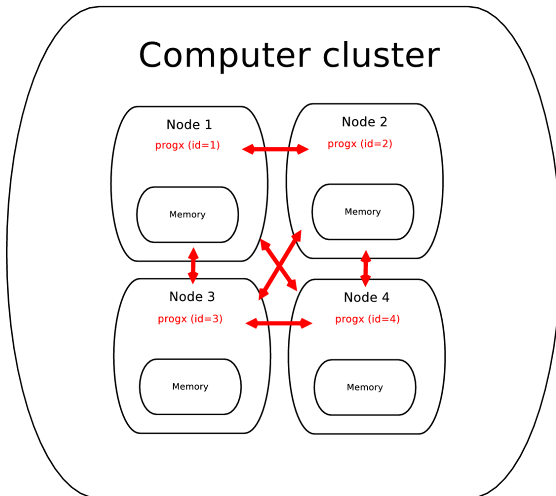
With help from the queueing system, a separate instance of the program is started on each node. Each instance is initialized with some information so that it can relate itself to the others, i.e. “I am process number 3”.

Distributed memory (III)



Distributed memory

Communication between processes



Message Passing Interface (MPI)

The Message Passing Interface (MPI) standard:

“MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.“

Open standard available on web:

`http://www.mcs.anl.gov/research/projects/mpi/`

MPI defines explicit API for message-passing between processes running on different nodes.

MPI

Explicit communication function calls

Example of sending message using MPI:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
int tid = pthread_self();
sprintf(message, "this message sent from worker %d, thread %d.", myid, tid);
int tag = send_task_info;
MPI_Status status;
int rc = MPI_Send(message, message_length, MPI_CHARACTER, 0, tag, parent);
```

Example of code for receiving messages using MPI:

```
int message_length = 100;
char message[message_length];
int tag = send_task_info;
MPI_Status status;
for (int i = 0; i < n_workers*2; ++i) {
  MPI_Recv(message, message_length, MPI_CHARACTER, MPI_ANY_SOURCE,  tag, everyone, &status);
  std::cout<<"Manager received worker message: "<<message<<std::endl;
}
```

MPI

Starting program via mpirun

An MPI program is started via the `mpirun` utility:

```
mpirun progx
```

Example of job script:

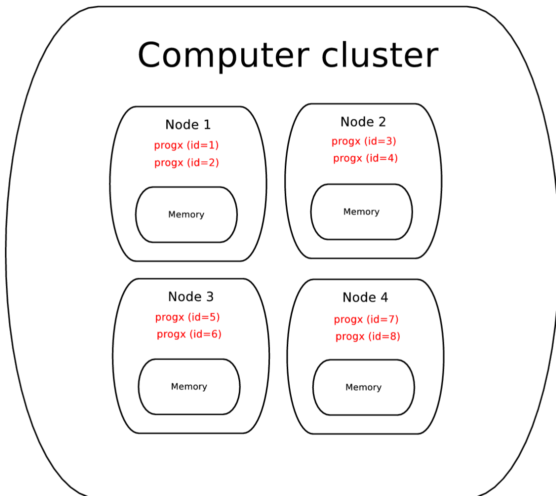
```
#!/bin/bash -l
#SBATCH -A g2011040
#SBATCH -p node -n 32
#SBATCH -t 01:00:00
#SBATCH -J mpitest
mpirun progx
```

The queueing system provides information to `mpirun` about how many processes of the program should be started, and on which nodes.

UPPSALA UNIVERSITET

MPI

One or several processes per node?



Distributed memory

Running jobs

An MPI program should be submitted to the queueing system as a "node" job, by using the following line in the job script:

```
#SBATCH -p node
```

Alternatively, you may specify `-p devel`.

For testing and debugging, you can also run MPI programs interactively. Then is can be convenient to test by running several MPI processes on the same node.

Hybrid approaches

When running on a compute cluster where each node is a multicore computer, it may be best to combine the two approaches: use MPI for communication between processes on different nodes, but use threads within each process to make best use of the available cores.

Parallel libraries

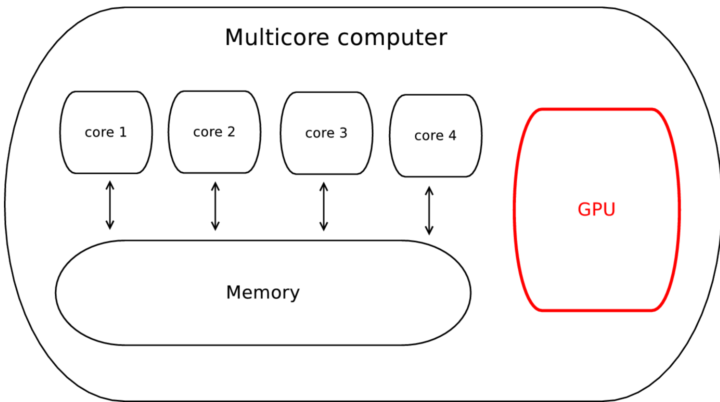
For matrix algebra operations on large dense matrices, there are parallel versions of BLAS and LAPACK: PBLAS and ScaLAPACK.

Outline

1. Parallelization strategies
2. Shared memory
3. Distributed memory
4. Parallelization using GPUs
5. Speedup and scaling

GPU:s

Graphics processing units (GPUs) can execute very large numbers of threads in parallel.



GPU:s

Limitations

Programming language alternatives:

- OpenCL
- Cuda

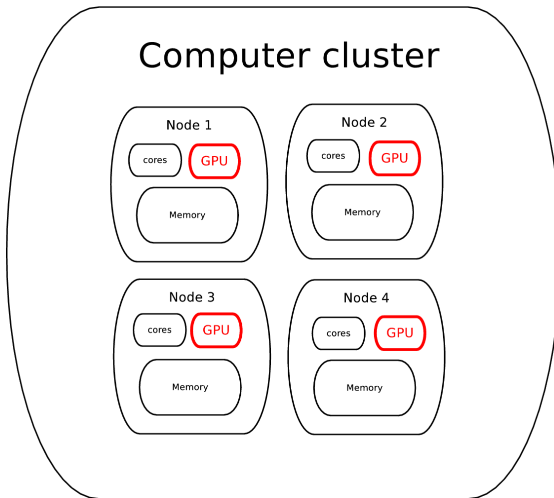
Major drawback of GPU parallelization: Tasks must be of (nearly) identical structure.

GPU:s

Linear algebra libraries

For dense matrix operations, the BLAS (and LAPACK?) routines are available for GPU:s, easy to use. With Cuda: “CudaBLAS”.

Clusters with GPU:s in each node



Programming for multicore+GPU clusters

On a compute cluster with GPU:s in each node, one can combine all three parallelization approaches:

- Use MPI for communication between processes on different nodes.
- Use threads to make use of the cores in each node.
- Use GPU:s to speedup the code where possible.

Outline

1 Parallelization strategies

2 Shared memory

3 Distributed memory

4 Parallelization using GPUs

5 Speedup and scaling

Speedup

The success of a parallelization effort is usually measured in terms of the achieved *speedup*; that is, how many times faster do you get the result compared to a serial program doing the same job.

$$S = \frac{t_{\mathrm{serial}}}{t_{\mathrm{parallel}}}$$

where t_{serial} and t_{parallel} are timings (wall time).

Speedup

Only trust the wall time!

Various ways of measuring “CPU time” exist, but can be misleading.

In general, best to stick to wall time measurements.

Scaling – weak vs strong

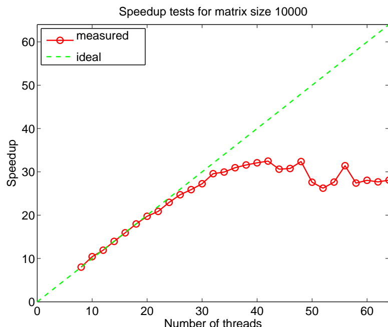
When talking about the "scaling" of a parallel implementation of a program, distinguish between *strong scaling* and *weak scaling*:

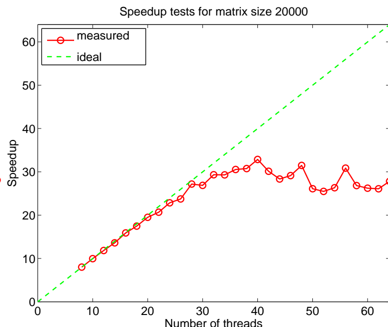
Strong scaling: run the program using varying numbers of threads/processes/workers, for the same problem size.

Weak scaling: run the program using varying numbers of threads/processes/workers, but scale up the problem size accordingly.

Scaling – weak vs strong

Example of strong scaling plots:





Trends for future clusters

At UPPMAX, the number of cores per node is increasing:

- 2007: Isis, 4 cores/node
- 2008: Grad, 8 cores/node
- 2010: Kalkyl, 8 cores/node
- 2011: New cluster, 16 cores/node

Same trend in other places, also for largest clusters in the world.

2020: 1024 cores/node??

Summary

Three kinds of parallelization

- Threading for multicore computers
- MPI for clusters
- OpenCL for GPU:s