COSC 6374
Parallel Computation

Data Parallel Approaches

Edgar Gabriel
Fall 2011

Data Parallel Approaches

- Parallelism with dependencies between tasks
  - Geometric domain decomposition
  - Pipeline pattern
- Parallelism with independent tasks
  - Loop level parallelism
  - Master-worker
  - Work-stealing
- Parallelism with limited dependencies between tasks
  - Divide and Conquer
  - MapReduce
Geometric domain decomposition

- All processes apply the same operations on different data items
- Key elements:
  - Data decomposition
  - Exchange and update operation
- No. of tasks = no. of processes/threads
- Processes and threads have to be executed simultaneously due to data dependencies
  - Communication or
  - Update of shared variables

Example: 2-D Laplace equation

- Parallel domain decomposition
  - Data exchange at process boundaries required
- Halo cells / Ghost cells
  - Copy of the last row/column of data from the neighbor process
Pipeline pattern

- Calculation can be viewed in terms of data flowing through a sequence of stages
- Computation performed on many data sets
  - Compare to pipelining in processors on the instruction level

```
<table>
<thead>
<tr>
<th>Pipeline stage 1</th>
<th>Pipeline stage 2</th>
<th>Pipeline stage 3</th>
<th>Pipeline stage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Pipeline pattern (II)

- Amount of concurrency limited to the number of stages of the pipeline
- Patterns works best, if amount of work performed by various stages is roughly equal
- Filling the pipeline: some stages will be idle
- Draining the pipeline: some stages will be idle
- Non-linear pipeline: pattern allows for different execution for different data items

```
<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3a</th>
<th>Stage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Pipeline pattern (III)

- Implementation:
  - Each stage typically assigned to a process/thread
  - A stage might be a data-parallel task itself
  - Computation per task has to be large enough to compensate for communication costs between the tasks

Loop level parallelism

- In many scientific applications, the most compute intensive part is organized in a large loop
- Splitting the loop execution onto different processes/threads is a straightforward parallelization,
- No dependencies between individual iterations of the loop allowed
- Can easily be mapped to OpenMP
  - But also possible in other programming models, e.g. POSIX Threads or MPI
Numerical integration - OpenMP example

```c
#include <stdio.h>
#include "omp.h"

int main ( int argc, char **argv )
{
    int i, num_steps=100000;
    double x, xn, pi, step, sum=0.0;
    step = (b-a)/num_steps;

#pragma omp parallel for private(x,xn) reduction(+:sum)
    for ( i=0; i<num_steps; i++) {
        x   = i * step;
        xn = (i+1) * step;
        sum = sum + 0.5*(xn-x)*(f(x)+f(xn);
    }
    return (0);
}
```

Numerical integration - MPI solution

```c
... int rank, size, i, num_steps=100000;
    double x, xn, end, step, sum, lsum=0.0;

    MPI_Init ( &argc, &argv );
    MPI_Comm_rank (MPI_COMM_WORLD, &rank );
    MPI_Comm_size (MPI_COMM_WORLD, &size );
    step  = (b-a)/num_steps;

    for ( i=rank; i<num_steps; i=i+size) {
        x   = i * step;
        xn = (i+1) * step;
        lsum = lsum + 0.5*(xn-x)*(f(x)+f(xn);
    }
    MPI_Allreduce (lsum, sum, 1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
    MPI_Finalize ();
...```
Dealing with independent tasks

- Some problem can be broken down into entirely independent tasks
- No. of tasks often much larger than number of processes/threads used
- Load balance - minimize the overall execution of all tasks
- Schedule: the way in which tasks are assigned to processes/threads for execution
- Two classes of scheduling approaches:
  - Static schedule: distribution of tasks to UEs is determined at the start of the computation and not changed anymore
  - Dynamic schedule: the distribution of tasks to UEs changes as the computation proceeds

Task scheduling - example

Independent tasks

Poor mapping to 4 UEs

Good mapping to 4 UEs
Static schedule

- Tasks are grouped into blocks of tasks
- Blocks are assigned to processors
- Each processor should take approximately same amount of time to complete task
- Static schedule usually used when
  - Availability of computational resources is predictable (e.g. dedicated usage of nodes)
  - Homogeneous hardware
  - Size of each task is nearly identical

Dynamic scheduling

- Used when
  - Effort associated with each task varies widely/is unpredictable
  - Heterogeneous hardware
- Common implementations:
  - usage of task queues: if a process/thread finishes current task, it removes the next task from the task-queue
- Trade-offs:
  - Fine grained (=shorter, smaller) tasks allow for better load balance
  - Fine grained task have higher costs for task management and dependency management
Master-Worker framework

Master Process
- Result queue
- Task queue

Worker Process 1

Worker Process 2

Master-Worker pattern (II)

- Two logically different entities: master process managing a work-queue, worker processes executing a task assigned to them by the master
- Completion: explicit notification of master to worker processes typically required
  - Can become very complicated for adaptive and recursive problems, where a worker can also ‘generate’ new tasks
- Master/worker pattern works well, if a master has sufficient worker processes
- Master process can become a bottleneck if tasks are too small and number of worker processes is very large
Master-Worker pattern (III)

- Useful if
  - Workload associated with tasks are highly variable - MW has ‘built-in’ load balancing
  - Hardware capabilities are non-uniform
  - Tasks are not tightly coupled - each worker process typically only has to communicate with the master process but not with other workers
- Relatively straight forward to provide fault-tolerance for the clients
  - Master process keeps track of who did what and can easily re-assign work of failed process

Master-Work MPI implementation (I)

```c
#define MASTERRANK 0
#define WORK_TAG         10
#define RES_TAG          11
#define NO_WORK_LEFT_TAG 12

int main ( int argc, char ** argv ) {  
  int rank, size, maxworkers;
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &rank );
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  maxworkers = size-1;
  if ( rank == MASTERRANK ) {  
    master(numworkers);
  }else {  
    worker();
  }  
  MPI_Finalize ();
  return (0);
}
```
Master-Work MPI implementation (II)

```c
int worker ( void )
{
    int done=0; /* condition set to false */
    MPI_Status status;
    while ( !done ) {
        MPI_Recv ( &work, maxcnt, MPI_DOUBLE, MASTERRANK,
                   MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if ( status.MPI_TAG == NO_WORK_LEFT_TAG ) {
            done = 1; /* condition set to true */
        }
        else {
            result = do_calculations ( work );
            MPI_Send (&result, rescnt, MPI_DOUBLE, MASTERRANK,
                      RES_TAG, MPI_COMM_WORLD);
        }
    }
    return (0);
}
```

Master-Work MPI implementation (III)

```c
int master ( int maxworkers )
{
    int done = 0, proc;

    /* distribute initial work */
    for ( proc=1; proc<maxworkers; proc++ ) {
        next = get_next_work ();
        MPI_Send ( &next, xx, MPI_DOUBLE, proc, WORK_TAG,
                   MPI_COMM_WORLD);
        marc_work_as_assigned ( proc, next );
    }

    while ( done < maxworkers ) {
        MPI_Recv ( &deltasres, xy, MPI_DOUBLE, MPI_ANY_SOURCE,
                   RES_TAG, comm, &status);
        proc = status.MPI_SOURCE;
        store_work_result ( proc, deltasres );
        next = get_next_work ();
    }
```
Master-Work MPI implementation (IV)

```c
if ( next != NO_WORK_LEFT ) {
    MPISend ( &next, xx, MPI_DOUBLE, proc, WORK_TAG,
             MPI_COMM_WORLD );
    marc_work_as_assigned ( proc, next );
} else {
    MPISend ( &next, 0, MPI_DOUBLE, j, NO_WORK_LEFT_TAG,
             MPI_COMM_WORLD );
    done ++;
}
/* end while loop */
return (0);
```

Task Parallelism using work stealing

- Work-stealing: each process/thread has its own work queue
- Once its queue is empty, a process/thread steals work from the task queue of another UE
Divide and Conquer algorithms

- A problem is split into a number of smaller sub-problems
- Each sub-problem is solved independently
- Sub-solutions of each sub-problem will be merged to the solution of the final problem
- Problems of Divide and Conquer for Parallel Computing:
  - Amount of exploitable concurrency decreases over the lifetime
  - Trivial parallel implementation: each function call to solve is a task on its own. For small problems, no new task should be generated, but the baseSolve should be applied
Divide and Conquer

- Implementation:
  - On shared memory machines, a divide and conquer algorithm can easily be mapped to a fork/join model
    - A new task is forked (=created)
    - After this task is done, it joins the original task (=destroyed)
  - On distributed memory machines: task queues
    - Often implemented using the Master/Worker framework - discussed later

```c
int solve ( Problem P )
{
    int solution;

    /* Check whether we can further partition the problem */
    if (baseCase(P)) {
        solution = baseSolve(P); /* No, we can’t */
    } else {
        /* yes, we can */
        Problem subproblems[N];
        int subsolutions[N];

        subproblems = split (P); /* Partition the problem */
        for ( i=0; i < N; i++ ) {
            subsolutions[i] = solve ( subproblems[i]);
        }
        solution = merge (subsolutions);
    }
    return ( solution );
}
```
MapReduce Programming Model

- Break down your algorithm in two parts:
  - ‘Embarrassingly parallel’ Map function:
    \[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]
    with key-value pairs \((K,V)\)
  - Reduce function:
    \[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]

Slide based on a lecture of Matei Zaharia: “Introduction to MapReduce and Hadoop”
http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt

Word Count Execution

Input: the quick brown fox
the fox ate the mouse
how now brown cow

Map: the, 1
brown, 1
fox, 1

Shuffle & Sort:
the, 1
brown, 1
fox, 1

Reduce:
brown, 2
fox, 2
how, 1
now, 1
the, 3

Output: the quick brown cow
ate, 1
cow, 1
mouse, 1
quick, 1

Slide based on a lecture of Matei Zaharia: “Introduction to MapReduce and Hadoop”
http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt
MapReduce Execution Details

- **Single master** controls job execution on multiple **workers** as well as user scheduling
- Mappers preferentially placed on same node or same rack as their input block
  - Push computation to data, minimize network use
- Mappers save outputs to local disk rather than pushing directly to reducers
  - Allows having more reducers than nodes
  - Allows recovery if a reducer crashes

---

An Optimization: The Combiner

- A combiner is a local aggregation function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases size of intermediate data

- Example: local counting for Word Count:

  ```python
  def combiner(key, values):
      output(key, sum(values))
  ```

---

Slide based on a lecture of Matei Zaharia: “Introduction to MapReduce and Hadoop”,
[http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt](http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt)
Parallelism in the reduction step

- Reduction operations are available in MPI and OpenMP
- Reduction step in MapReduce can be more general than what MPI and OpenMP offer
  - E.g. provided a list of words and the number of counts of each keyword, reduction step could provided the 20 most utilized keywords
  - Parallelism in reduction operation still often available
MapReduce - why all the hype?

- Pattern can be applied to a significant number of applications
  - But not all!
- Existing solutions (e.g. Hadoop) considered easier to use and deploy than generic parallel computing solutions
- Built-in fault tolerance often available for existing solutions

References

More information on MapReduce:
- Matei Zaharia: “Introduction to MapReduce and Hadoop”, http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/Cloud_MapReduce_Zaharia.ppt