Basic Concepts in Parallelization

Ruud van der Pas

Senior Staff Engineer
Oracle Solaris Studio
Oracle
Menlo Park, CA, USA

IWOMP 2010
CCS, University of Tsukuba
Tsukuba, Japan
June 14-16, 2010
Outline

- Introduction
- Parallel Architectures
- Parallel Programming Models
- Data Races
- Summary
Introduction
Why Parallelization?

Parallelization is another optimization technique. The goal is to reduce the execution time.

To this end, multiple processors, or cores, are used.

Using 4 cores, the execution time is 1/4 of the single core time.
What Is Parallelization?

"Something" is parallel if there is a certain level of independence in the order of operations.

In other words, it doesn't matter in what order those operations are performed.

- A sequence of machine instructions
- A collection of program statements
- An algorithm
- The problem you're trying to solve
What is a Thread?

- Loosely said, a thread consists of a series of instructions with its own program counter ("PC") and state
- A parallel program executes threads in parallel
- These threads are then scheduled onto processors
Parallel overhead

- The total CPU time often exceeds the serial CPU time:
  - The newly introduced parallel portions in your program need to be executed
  - Threads need time sending data to each other and synchronizing (“communication”)
    - Often the key contributor, spoiling all the fun
- Typically, things also get worse when increasing the number of threads
- Efficient parallelization is about minimizing the communication overhead
Communication

Wallclock time

Serial Execution

Parallel - Without communication

- Embarrassingly parallel: 4x faster
- Wallclock time is ¼ of serial wallclock time

Parallel - With communication

- Additional communication
- Less than 4x faster
- Consumes additional resources
- Wallclock time is more than ¼ of serial wallclock time
- Total CPU time increases
Load balancing

Perfect Load Balancing

- All threads finish in the same amount of time
- No threads is idle

Load Imbalance

- Different threads need a different amount of time to finish their task
- Total wall clock time increases
- Program does not scale well
About scalability

- Define the speed-up $S(P)$ as $S(P) := T(1)/T(P)$
- The efficiency $E(P)$ is defined as $E(P) := S(P)/P$
- In the ideal case, $S(P) = P$ and $E(P) = P/P = 1 = 100\%$
- Unless the application is embarrassingly parallel, $S(P)$ eventually starts to deviate from the ideal curve
- Past this point $P_{opt}$, the application sees less and less benefit from adding processors
- Note that both metrics give no information on the actual run-time
- As such, they can be dangerous to use

In some cases, $S(P)$ exceeds $P$

This is called "superlinear" behaviour

Don't count on this to happen though
Amdahl's Law

Assume our program has a parallel fraction “f”

This implies the execution time $T(1) := f \cdot T(1) + (1-f) \cdot T(1)$

On P processors: $T(P) = (f/P) \cdot T(1) + (1-f) \cdot T(1)$

Amdahl's law:

$$S(P) = T(1)/T(P) = 1 / (f/P + 1-f)$$

Comments:

- This "law' describes the effect the non-parallelizable part of a program has on scalability
- Note that the additional overhead caused by parallelization and speed-up because of cache effects are not taken into account
Amdahl's Law

- It is easy to scale on a small number of processors
- Scalable performance however requires a high degree of parallelization i.e. f is very close to 1
- This implies that you need to parallelize that part of the code where the majority of the time is spent
Amdahl's Law in practice

We can estimate the parallel fraction “f”

Recall: \( T(P) = (f/P)\cdot T(1) + (1-f)\cdot T(1) \)

It is trivial to solve this equation for “f”:

\[
f = \frac{1 - \frac{T(P)}{T(1)}}{1 - \frac{1}{P}}
\]

Example:

\[
T(1) = 100 \text{ and } T(4) = 37 \implies S(4) = \frac{T(1)}{T(4)} = 2.70
\]

\[
f = \frac{1 - 37/100}{1 - (1/4)} = \frac{0.63}{0.75} = 0.84 = 84\%
\]

Estimated performance on 8 processors is then:

\[
T(8) = \left(\frac{0.84}{8}\right)\cdot 100 + \left(1 - 0.84\right)\cdot 100 = 26.5
\]

\[
S(8) = \frac{T}{T(8)} = 3.78
\]
Threads Are Getting Cheap.....

Elapsed time and speed up for f = 84%

Using 8 cores:
100/26.5 = 3.8x faster

Number of cores used

= Elapsed time
= Speed up
Numerical results

Consider:

\[ A = B + C + D + E \]

Serial Processing

- \[ A = B + C \]
- \[ A = A + D \]
- \[ A = A + E \]

Parallel Processing

Thread 0

- \[ T1 = B + C \]
- \[ T1 = T1 + T2 \]

Thread 1

- \[ T2 = D + E \]

☞ The roundoff behaviour is different and so the numerical results may be different too

☞ This is natural for parallel programs, but it may be hard to differentiate it from an ordinary bug ....
Cache Coherence
Typical cache based system
Cache line modifications

- Page
- Cache line
- Caches

Main Memory

Cache line inconsistency!
Caches in a parallel computer

- A cache line starts in memory
- Over time multiple copies of this line may exist

Cache Coherence ("cc"):
- Tracks changes in copies
- Makes sure correct cache line is used
- Different implementations possible
- Need hardware support to make it efficient
Parallel Architectures
Uniform Memory Access (UMA)

- Also called "SMP" (Symmetric Multi Processor)
- Memory Access time is Uniform for all CPUs
- CPU can be multicore
- Interconnect is "cc":
  - Bus
  - Crossbar
- No fragmentation - Memory and I/O are shared resources

Pro
- Easy to use and to administer
- Efficient use of resources

Con
- Said to be expensive
- Said to be non-scalable
NUMA

- Also called "Distributed Memory" or NORMA (No Remote Memory Access)
- Memory Access time is Non-Uniform
- Hence the name "NUMA"
- Interconnect is not "cc": Ethernet, Infiniband, etc, ......
- Runs 'N' copies of the OS
- Memory and I/O are distributed resources

Pro
- Said to be cheap
- Said to be scalable

Con
- Difficult to use and administer
- Inefficient use of resources
The Hybrid Architecture

- Second-level interconnect is not cache coherent
  - *Ethernet, Infiniband, etc, ....*

- Hybrid Architecture with all Pros and Cons:
  - *UMA within one SMP/Multicore node*
  - *NUMA across nodes*
cc-NUMA

- Two-level interconnect:
  - UMA/SMP within one system
  - NUMA between the systems

- Both interconnects support cache coherence i.e. the system is fully cache coherent

- Has all the advantages ('look and feel') of an SMP

- Downside is the Non-Uniform Memory Access time
Parallel Programming Models
How To Program A Parallel Computer?

- There are numerous parallel programming models
- The ones most well-known are:
  - **Distributed Memory**
    - Sockets (standardized, low level)
    - PVM - Parallel Virtual Machine (obsolete)
    - MPI - Message Passing Interface (de-facto std)
  - **Shared Memory**
    - Posix Threads (standardized, low level)
    - OpenMP (de-facto standard)
    - Automatic Parallelization (compiler does it for you)
Parallel Programming Models
Distributed Memory - MPI
What is MPI?

- **MPI stands for the “Message Passing Interface”**
- **MPI is a very extensive de-facto parallel programming API for distributed memory systems (i.e. a cluster)**
  - An MPI program can however also be executed on a shared memory system
- **First specification: 1994**
- **The current MPI-2 specification was released in 1997**
  - **Major enhancements**
    - Remote memory management
    - Parallel I/O
    - Dynamic process management
More about MPI

- **MPI has its own data types (e.g. MPI_INT)**
  - User defined data types are supported as well
- **MPI supports C, C++, and Fortran**
  - Include file `<mpi.h>` in C/C++ and `"mpif.h"` in Fortran
- **An MPI environment typically consists of at least:**
  - A library implementing the API
  - A compiler and linker that support the library
  - A runtime environment to launch an MPI program
- **Various implementations available**
  - HPC Clustertools, MPICH, MVAPICH, LAM, Voltaire MPI, Scali MPI, HP-MPI, .....
The MPI Programming Model

A Cluster Of Systems
The MPI Execution Model

All processes start

MPI_Init

= communication

MPI_Finalize

All processes finish
The MPI Memory Model

- All threads/processes have access to their own, private, memory only
- Data transfer and most synchronization has to be programmed explicitly
- All data is private
- Data is shared explicitly by exchanging buffers
Example - Send “N” Integers

```c
#include <mpi.h>  

you = 0; him = 1;  

MPI_Init(&argc, &argv);  

MPI_Comm_rank(MPI_COMM_WORLD, &me);  

if ( me == 0 ) {  
    error_code = MPI_Send(&data_buffer, N, MPI_INT,  
                           him, 1957, MPI_COMM_WORLD);  
} else if ( me == 1 ) {  
    error_code = MPI_Recv(&data_buffer, N, MPI_INT,  
                           you, 1957, MPI_COMM_WORLD,  
                           MPI_STATUS_IGNORE);  
}

MPI_Finalize();  
```
Run time Behavior

**Process 0**
- $you = 1$
- $him = 0$
- $me = 0$
- **MPI_Send**

**Process 1**
- $you = 1$
- $him = 0$
- $me = 1$

---

N integers
destination = $you = 1$
label = 1957

---

N integers
sender = $him = 0$
label = 1957

---

Yes! Connection established
The Pros and Cons of MPI

- **Advantages of MPI:**
  - **Flexibility** - Can use any cluster of any size
  - **Straightforward** - Just plug in the MPI calls
  - **Widely available** - Several implementations out there
  - **Widely used** - Very popular programming model

- **Disadvantages of MPI:**
  - **Redesign of application** - Could be a lot of work
  - **Easy to make mistakes** - Many details to handle
  - **Hard to debug** - Need to dig into underlying system
  - **More resources** - Typically, more memory is needed
  - **Special care** - Input/Output
Parallel Programming Models
Shared Memory - Automatic Parallelization
Automatic Parallelization (-xautopar)

- Compiler performs the parallelization (loop based)
- Different iterations of the loop executed in parallel
- *Same* binary used for *any* number of threads

```c
for (i=0; i<1000; i++)
a[i] = b[i] + c[i];
```

```
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-249</td>
<td>250-499</td>
<td>500-749</td>
<td>750-999</td>
</tr>
</tbody>
</table>
```

- `OMP_NUM_THREADS=4`
Parallel Programming Models
Shared Memory - OpenMP
**OpenMP**

http://www.openmp.org
Shared Memory Model

Programming Model

- All threads have access to the same, **globally shared**, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit
About data

- In a shared memory parallel program variables have a "label" attached to them:

  - Labelled "Private"  ➙ Visible to one thread only
    - Change made in local data, is not seen by others
    - Example - Local variables in a function that is executed in parallel

  - Labelled "Shared"  ➙ Visible to all threads
    - Change made in global data, is seen by all others
    - Example - Global data
Example - Matrix times vector

```c
#pragma omp parallel for default(none) \ 
private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
    {
        sum += b[i][j]*c[j];
        a[i] = sum;
    }
}
```

```
TID = 0
```

```
for (i=0,1,2,3,4)
    
    \[ i = 0 \]
    sum = b[i=0][j]*c[j]
    a[0] = sum

    \[ i = 1 \]
    sum = b[i=1][j]*c[j]
    a[1] = sum

... etc ...
```

```
TID = 1
```

```
for (i=5,6,7,8,9)
    
    \[ i = 5 \]
    sum = b[i=5][j]*c[j]
    a[5] = sum

    \[ i = 6 \]
    sum = b[i=6][j]*c[j]
    a[6] = sum

... etc ...
```
## A Black and White comparison

### MPI
- De-facto standard
- Endorsed by all key players
- Runs on any number of (cheap) systems
- “Grid Ready”
- High and steep learning curve
- You're on your own
- All or nothing model
- No data scoping (shared, private, ..)
- More widely used (but ....)
- Sequential version is not preserved
- Requires a library only
- Requires a run-time environment
- Easier to understand performance

### OpenMP
- De-facto standard
- Endorsed by all key players
- Limited to one (SMP) system
- Not (yet?) “Grid Ready”
- Easier to get started (but, ...)
- Assistance from compiler
- Mix and match model
- Requires data scoping
- Increasingly popular (CMT !)
- Preserves sequential code
- Need a compiler
- No special environment
- Performance issues implicit
The Hybrid Parallel Programming Model
The Hybrid Programming Model

Distributed Memory

Shared Memory

0 1 P
M M M

Shared Memory

0 1 P
M M M
Data Races
About Parallelism

- **Parallelism**
- **Independence**
- **No Fixed Ordering**

"Something" that does not obey this rule, is not parallel (at that level ...)

Basic Concepts in Parallelization

Tutorial IWOMP 2010 – CCS Un. of Tsukuba, June 14, 2010
Shared Memory Programming

Threads communicate via shared memory
What is a Data Race?

- Two different threads in a multi-threaded shared memory program
- Access the same (=shared) memory location
  - Asynchronously
  - Without holding any common exclusive locks
  - At least one of the accesses is a write/store
Example of a data race

```c
#pragma omp parallel shared(n)
{n = omp_get_thread_num();}
```
Another example

```c
#pragma omp parallel shared(x)
{x = x + 1;}
```
About Data Races

- Loosely described, a data race means that the update of a shared variable is not well protected.

- A data race tends to show up in a nasty way:
  - Numerical results are (somewhat) different from run to run.
  - Especially with Floating-Point data difficult to distinguish from a numerical side-effect.
  - Changing the number of threads can cause the problem to seemingly (dis)appear.
    
    ✔ May also depend on the load on the system.
  - May only show up using many threads.
A parallel loop

```c
for (i=0; i<8; i++)
    a[i] = a[i] + b[i];
```

Every iteration in this loop is independent of the other iterations.

Thread 1

- `a[0] = a[0] + b[0]`

Thread 2

Not a parallel loop

```c
for (i=0; i<8; i++)
a[i] = a[i+1] + b[i];
```

The result is not deterministic when run in parallel!

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>

Time
About the experiment

- We manually parallelized the previous loop
  - The compiler detects the data dependence and does not parallelize the loop
- Vectors $a$ and $b$ are of type integer
- We use the checksum of $a$ as a measure for correctness:
  - $\text{checksum} += a[i]$ for $i = 0, 1, 2, \ldots, n-2$
- The correct, sequential, checksum result is computed as a reference
- We ran the program using 1, 2, 4, 32 and 48 threads
  - Each of these experiments was repeated 4 times
## Numerical results

<table>
<thead>
<tr>
<th>threads: 1 checksum</th>
<th>correct value</th>
<th>1953</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads: 1 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 1 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 1 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threads: 2 checksum</th>
<th>correct value</th>
<th>1953</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads: 2 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 2 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 2 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threads: 4 checksum</th>
<th>correct value</th>
<th>1953</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads: 4 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 4 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 4 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threads: 32 checksum</th>
<th>correct value</th>
<th>1953</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads: 32 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 32 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 32 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threads: 48 checksum</th>
<th>correct value</th>
<th>1953</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads: 48 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 48 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
<tr>
<td>threads: 48 checksum</td>
<td>correct value</td>
<td>1953</td>
</tr>
</tbody>
</table>

## Data Race In Action!
Summary
Parallelism Is Everywhere

Multiple levels of parallelism:

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Technology</th>
<th>Programming Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Level</td>
<td>Superscalar</td>
<td>Compiler</td>
</tr>
<tr>
<td>Chip Level</td>
<td>Multicore</td>
<td>Compiler, OpenMP, MPI</td>
</tr>
<tr>
<td>System Level</td>
<td>SMP/cc-NUMA</td>
<td>Compiler, OpenMP, MPI</td>
</tr>
<tr>
<td>Grid Level</td>
<td>Cluster</td>
<td>MPI</td>
</tr>
</tbody>
</table>

Threads Are Getting Cheap