Efficient OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS

Woo-Chul Jeun and Soonhoi Ha
Seoul National University, Korea
2007.01.24
wcjeun@iris.snu.ac.kr
Contents

- Background

- OpenMP implementation for MPSoC configurations

- Our OpenMP implementation and translation for a target platform

- Conclusions & Future directions
Background
MPSoC
(Multiprocessor System-on-Chip)

- Attractive platform for computation-intensive applications such as multimedia encoder/decoder

- Can have various architectures according to target applications
  - Memory: Shared, Distributed, and Hybrid
  - OS: SMP(symmetric multiprocessor) kernel Linux, small operating systems, and no operating system

- No standard parallel programming model for MPSoC

- Not sufficient studies on various MPSoC architectures
## Parallel Programming Models

<table>
<thead>
<tr>
<th>Memory</th>
<th>Distributed memory (private memory)</th>
<th>Shared memory (same address space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm.</td>
<td>Explicit message-passing</td>
<td>Memory access</td>
</tr>
<tr>
<td>Programming easiness</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td>Performance</td>
<td>Optimize data distribution, data transfer, and synchronization manually</td>
<td>Complicate issues depend on OpenMP implementation</td>
</tr>
</tbody>
</table>

- Manual optimization (MPI) vs. Easy programming (OpenMP)
OpenMP Overview

- **Specification standard** to represent programmer’s intention with compiler directives (C/C++ and Fortran)
- OpenMP is an attractive parallel programming model for MPSoC because of easy programming.
- Programmers can write an OpenMP program by inserting OpenMP directives into a serial program.

```c
#pragma omp parallel for
for( i = 0 ; i < 1000 ; i++ ) {
    data[i] = 0;
}
```
OpenMP execution model: fork/join

```c
#pragma omp parallel for
for (i = 0; i < 1000; i++) {
    data[i] = 0;
}
```

All threads divide and execute its own computation workload.
OpenMP programming environment

- **OpenMP** does not define how to implement OpenMP directives on a parallel processing platform.

- OpenMP runtime system : OpenMP directive implementation with libraries on a target platform

- OpenMP translator (for C language) : converts an OpenMP program into the codes (C codes) using the OpenMP runtime system
Hybrid execution model

- Separating parallel programming model from execution model

- Application programmers use OpenMP.

- OpenMP runtime system can use hybrid execution model of message passing model and shared address space model for the performance improvement.

- Easy programming and High performance
Motivation

- OpenMP runtime system depends on a target platform.

- OpenMP translator is customized to the OpenMP runtime system.

- To get high performance on various platforms, it is necessary to research on efficient OpenMP runtime system and OpenMP translator for each target platform.
### Terminologies

**Barrier synchronization**
- Every processor (or thread) waits until all processors (or threads) arrive at a synchronization point.

**Reduction operation**
- Integrates operations of all processors (or threads) into one operation.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum: 0</td>
<td>Sum: 0</td>
<td>Sum: 0</td>
</tr>
<tr>
<td><code>sum+=a;</code></td>
<td><code>sum+=b;</code></td>
<td><code>sum+=c;</code></td>
</tr>
<tr>
<td>Reduction operation</td>
<td>Reduction operation</td>
<td>Reduction operation</td>
</tr>
<tr>
<td><code>sum+=(a+b+c);</code></td>
<td><code>sum+=(a+b+c);</code></td>
<td><code>sum+=(a+b+c);</code></td>
</tr>
<tr>
<td>Sum: a+b+c</td>
<td>Sum: a+b+c</td>
<td>Sum: a+b+c</td>
</tr>
</tbody>
</table>
OpenMP implementation on MPSoC configurations
### Possible OpenMP implementation on MPSoC configurations

<table>
<thead>
<tr>
<th>OS with thread library</th>
<th>Distributed memory</th>
<th>Shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread programming</strong> + SDSM : fault handler</td>
<td><strong>Thread programming</strong> + Shared memory (Yoshihiko et al.)</td>
<td></td>
</tr>
</tbody>
</table>

| Without OS | Processor programming + SDSM : message passing | Processor programming + Shared memory (Feng Liu et al., Ours) |

- SDSM (software distributed shared memory) vs. Shared memory
- Thread programming vs. Processor programming
Shared memory + OS with thread library

- Distributed memory
- Thread programming + SDSM: fault handler
- Processor programming + SDSM: message passing

OS with thread library

Without OS

- No need for memory consistency protocol
- Similar to thread programming in a SMP machine (Ex. dual processor PC)
- Yoshihiko Hotta et al., [EWOMP’2004]
  - SMP (symmetric multiprocessor) kernel Linux and POSIX thread library
  - Similar to OpenMP implementation and translation in a SMP machine (dual processor PC)
  - They focused on power optimization
Shared memory + No OS

- No need for memory consistency protocol
- Make processors run a program in parallel (load and initiate processors)
- Feng Liu et al., [WOMPAT’03, ICPP’2003]
  - No operating system
  - Their own OpenMP directive extension for DSP
  - OpenMP directive extension for special hardware on CT3400
  - Harmful barrier synchronization implementation
Our OpenMP implementation and translation on a target multiprocessor system-on-chip platform
CT3400 Architecture
(Cradle Technologies, Inc.)

- 230MHz processor, hardware semaphores (32 local, 64 global)
- Shared memory; No operating system and no thread library

- RISC-like processor
- Instruction cache (32KB)
- Global bus interface
- Off-chip memory (256MB)
- Global semaphores
- On-chip memory (64KB)
- Local instruction bus
- Local data bus
- Global bus
- Semaphores

Instruction cache (32KB)
Initialization

OpenMP translator extracts original main function to a function. (app_main())
- Make new main function call original main
- Initialization procedure loads program codes on other processors and initiates them before application starts (initializer())

Original main

```c
int main(...) {
    {  
    }
}
```

OpenMP translation

```c
int main(...) {
    initializer(...);
    app_main(...);
}
```

New main on master node

```c
int app_main(...) {
}
```
Parallelization

OpenMP translator extracts a parallel region to a function

- All processors execute the function. (cf. thread)
- Master processor executes serial region and other processors wait until master processor arrives at a parallel region.

```c
#pragma omp parallel for
for(i=0;i<1000;i++){
    data[i]=0;
}
```

New main

```c
parallelize(…, parallel_region_0)
```

```c
void parallel_region_0(…) {
    for(i=start;i<end;i++){
        data[i]=0;
    }
}
```
Translation of global shared variables

- ‘cragcc’ C compiler on CT3400 can process global variables efficiently
- OpenMP translator can translate global shared variables with two memory allocation methods.
  - **Static allocation**
    - `int data[100];`, global data area (0%~31% better)
    - OpenMP translator can inform the OpenMP runtime system that the variables are in global data area.
  - **Dynamic allocation**
    - `int *data; data = allocate_local(...);`, heap area
    - OpenMP runtime system cannot know whether the variables are global variables.
### 24*24 Matrix multiplication (cycles)

- On the cycle-accurate simulator “Inspector” provided by Cradle technologies, Inc.

<table>
<thead>
<tr>
<th>Processes</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>3,664,513</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Parallel (hand-written)</td>
<td>3,653,761</td>
<td>1,827,537</td>
<td>914,127</td>
</tr>
<tr>
<td>OpenMP, Dynamic</td>
<td>5,221,225</td>
<td>2,622,901</td>
<td>1,320,474</td>
</tr>
<tr>
<td>OpenMP, Static</td>
<td>3,674,336</td>
<td>1,845,050</td>
<td>933,549</td>
</tr>
</tbody>
</table>

- Static memory allocation is 31% better than dynamic memory allocation on CT3400.
Reduction (using temporary variable)

- Uses a temporary variable. (temp_var)
- Similar to thread programming
- Each processor updates the temporary variable with semaphore.
- All operations are serialized.
Reduction
(using temporary buffer array)

- Uses a temporary buffer array (buffer).
- Each processor updates its own element of the array without semaphore.
- All operations can be executed in parallel.
EPCC OpenMP micro-benchmark (cycles)

- On the cycle-accurate simulator “Inspector” provided by Cradle technologies, Inc.

<table>
<thead>
<tr>
<th>Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction, temporary variable</td>
<td>1,713</td>
<td>8,790</td>
<td>14,028</td>
</tr>
<tr>
<td>Reduction, temporary buffer array</td>
<td>1,713</td>
<td>7,805</td>
<td>12,631</td>
</tr>
</tbody>
</table>

- Temporary buffer array method is 10% better than temporary variable method on CT3400.
Previous harmful barrier synchronization implementation (example error case)

1. semaphore_lock(Sem.p);
2. done_pe++;
3. semaphore_unlock(Sem.p);
4. while( done_pe < PES )
   _pe_delay(1);
5. if( my_peid == 0 )
   done_pe = 0;

- **PES (Number of processors)**: 2
- **done_pe** (counter variable for synchronization)
- **my_peid** (processor ID)
- Processor 0 increases ‘done_pe’ to 1 with semaphore and does busy waiting.
Previous harmful barrier synchronization implementation (example error case)

1. semaphore_lock(Sem.p);
2. done_pe++;
3. semaphore_unlock(Sem.p);
4. while( done_pe < PES )
5.   _pe_delay(1);
6. if( my_peid == 0 )
7.   done_pe = 0;

- Processor 1 increases the counter.
- Processor 0 can exit the busy waiting loop.
Previous harmful barrier synchronization implementation (example error case)

Processor 0
semaphore_lock(Sem.p);
done_pe++;
semaphore_unlock(Sem.p);
while( done_pe < PES )
    _pe_delay(1);
if( my_peid == 0 )
done_pe = 0;

Processor 1

( done_pe : 0 , PES : 2 )

Processor 0 initializes the counter variable before processor 1 checks the value of the counter variable.
Previous harmful barrier synchronization implementation (example error case)

```c
1   semaphore_lock(Sem.p);
2   done_pe++;
3   semaphore_unlock(Sem.p);
4   while( done_pe < PES )
5     _pe_delay(1);
6   if( my_peid == 0 )
7     done_pe = 0;
```

- **Processor 1** cannot exit the loop and it fails for synchronization (**wrong**)
- **Wrong assumption of this implementation** : last processor is always processor 0 and it initializes the counter of current barrier
Our barrier implementation

- Introduce a phase variable and toggle phase of barrier to discriminate consequent barriers
- Initialize the counter of next barrier

```
1  semaphore_lock(Sem.p);
2  done_pe++;
3  semaphore_unlock(Sem.p);
4  while( done_pe < PES )
5     _pe_delay(1);
6  if( my_peid == 0 )
7     done_pe = 0;
```

```
1  semaphore_lock(Sem.p);
2  phase = (phase + 1) % 2;
3  if( done_pe[phase] + 1 == PES )
4      done_pe[(phase + 1) % 2] = 0;
5  done_pe[phase]++;
6  semaphore_unlock(Sem.p);
7  while( done_pe[phase] < PES )
8     _pe_delay(1);
```
**Our barrier implementation**

```c
1    semaphore_lock(Sem.p);
2    phase = (phase + 1) % 2;
3    if( done_pe[phase] + 1 == PES )
4        done_pe[( phase + 1 ) % 2 ] = 0;
5    done_pe[phase]++;
6    semaphore_unlock(Sem.p);
7    while( done_pe[phase] < PES )
8        _pe_delay(1);
```

- **Initialize the counter variable of next barrier and keep the counter variable of current barrier at the same time.**
Conclusions & Future directions

- When we translate global shared variables, static memory allocation is 31% better than dynamic memory allocation.
- For the reduction implementation, temporary buffer array method is 10% better than temporary variable method.
- We fixed previous harmful barrier synchronization implementation.
- Future directions
  - MPSoC with Distributed Memory
  - MPSoC with Heterogeneous processors (Ex. DSP)