1 OpenMP for Chip Multiprocessors[†]

Feng Liu and Vipin Chaudhary
Institute for Scientific Computing
Wayne State University
Detroit, MI 48202
{fliu, vipin}@wayne.edu

1.1 INTRODUCTION

Modern System-on-Chip (SOC) design shows a clear trend towards integration of multiple processor cores, the SOC System Section of the "International Technology Roadmap for Semiconductors" (http://public.itrs.net/) predicts that the number of processor cores will increase dramatically to match the processing demands of future applications. Providers like Intel, IBM, TI, Motorola, and Cradle have addressed own approaches for their high volume markets. Prior chip-level SOCs have been proposed using multiple copies of the same core (i.e., homogeneous), while core diversity offers better performance and greater ability to adapt to the demands of applications.

Developing a standard programming paradigm for parallel machines has been a major objective in parallel software research. Such standardization would not only facilitate the portability of parallel programs, but reduce the burden of parallel programming as well. Two major models for parallel machines are clusters or distributed memory machines and Symmetric Multiprocessor machines (SMP). Several parallel programming standards have been developed for individual architecture, such as the Message-Passing Interface (MPI) for distributed memory machines, and OpenMP or thread libraries (i.e. Pthread) for shared memory machines.

Chip Multiprocessors become an emerging parallel machine architecture. Currently there are no programming standards for SOCs or heterogeneous chip multiprocessors. Developers are required to write complex assembly language and/or C programs for SOCs. It's beneficial to incorporate high-level standardization like

[†] **This research was supported in part by Institute for Manufacturing Research and Institute for Scientific Computing, Wayne State University.**

OpenMP to improve program effectiveness. OpenMP is an industrial standard for shared memory parallel programming agreed on by a consortium of software and hardware vendors [1, 2]. It consists of a collection of compiler directives, library routines, and environment variables that can be easily inserted into a sequential program to create a portable program that will run in parallel on shared memory architectures.

In this chapter, we propose extensions to OpenMP to deal with the heterogeneity of chip multiprocessors. The heterogeneity is an important feature for most chip multiprocessors in the embedded space. Typical SOCs incorporate different types of processors into one die, i.e. RISC, or DSP-like processors. For parallel chips like *Software Scalable System on Chip (3SoC)* from Cradle, the parallelism is divided among processors; each processor may have different instruction set. By extending OpenMP, we can deploy different types of processors for parallel programming. We also focus on extending OpenMP for optimization. Our implementation of OpenMP compiler shows that OpenMP extensions can be used for optimization of parallel programs on chip multiprocessors architecture. The current version of our compiler accepts standard OpenMP programs and our extensions to OpenMP. Our performance evaluation shows scalable speedup using different types of processors and performance improvement through individual optimization extension on 3SoC.

The rest of this chapter is organized into eight sections. Section 1.2 introduces the 3SoC architecture. In Section 1.3, we discuss our compiler/translator for chip multiprocessor. Section 1.4 describes our extensions to OpenMP to deal with the heterogeneity. Optimization techniques to improve OpenMP performance on CMP are discussed in Section 1.5. Section 1.6 discusses the general implementation of this compiler. Performance evaluation and results are showed in Section 1.7. Finally, we summarize our conclusion in Section 1.8.

1.2 3SOC ARCHITECTURE OVERVIEW

Cradle's *Software Scalable System on Chip (3SoC)* architecture consists of dozens of high performance RISC-like and digital signal processors on a single chip with fully software programmable and dedicated input-output processors. The processors are organized into small groups, with eight digital signal processors and four RISC-like processors each sharing a block of local data and control memory, with all groups having access to global information via a unique on-chip bus-the Global Bus. It is because data, signal, and I/O processors are all available on a single chip, and that the chip is thereby capable of implementing entire systems [3]. The block diagram is shown as Fig. 1.1.

The 3SoC is a shared memory MIMD (multiple instruction/multiple data) computer that uses a single 32-bit address space for all register and memory elements. Each register and memory element in the 3SoC has a unique address and is uniquely addressable.

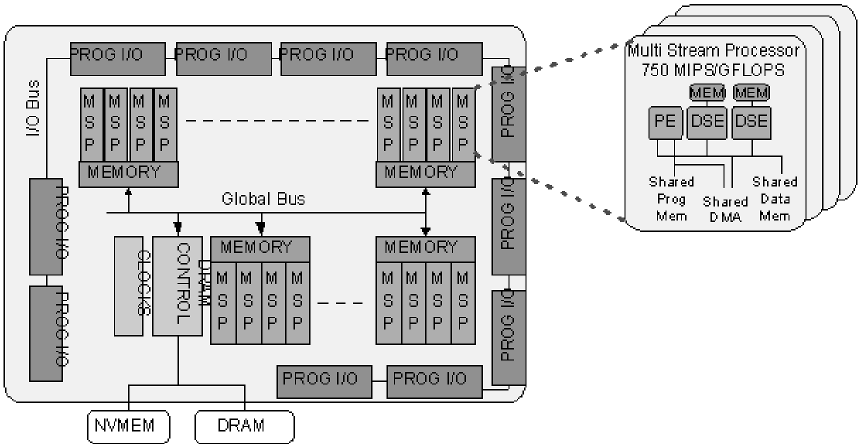


Fig. 1.1 3SoC Block Diagram.

1.2.1 Quads

The Quad is the primary unit of replication for 3SoC. A 3SoC chip has one or more Quads, with each Quad consisting of four PEs, eight DSEs, and one Memory Transfer Engine (MTE) with four Memory Transfer Controllers (MTCs). In addition, PEs share 32KB of instruction cache and Quads share 64KB of data memory, 32K of which can be optionally configured as cache. Thirty-two semaphore registers within each quad provide the synchronization mechanism between processors. Fig. 1.2 shows a Quad block diagram. Note that the Media Stream Processor (MSP) is a logical unit consisting of one PE and two DSEs.

- **Processing Element** : The PE is a 32-bit processor with 16-bit instructions and thirty-two 32-bit registers. The PE has a RISC-like instruction set consisting of both integer and IEEE 754 floating point instructions. The instructions have a variety of addressing modes for efficient use of memory. The PE is rated at approximately 90 MIPS.

- **Digital Signal Engine** : The DSE is a 32-bit processor with 128 registers and local program memory of 512 20-bit instructions optimized for high-speed fixed and floating point processing. It uses MTCs in the background to transfer data between the DRAM and the local memory. The DSE is the primary compute engine and is rated at approximately 350 MIPS for integer or floating-point performance.

1.2.2 Communication and Synchronization

- **Communication** : Each Quad has two 64-bit local buses: an instruction bus and a data bus. The instruction bus connects the PEs and MTE to the

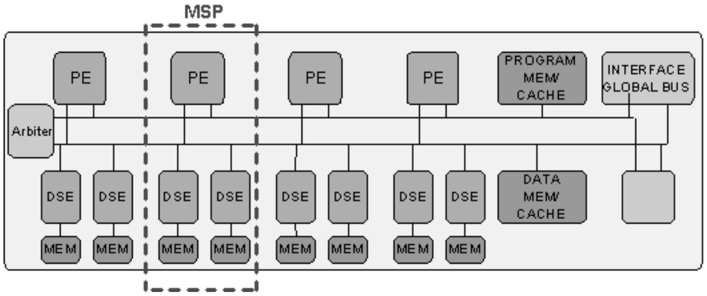


Fig. 1.2 Quad Block diagram.

instruction cache. The data bus connects the PEs, DSEs, and MTE to the local data memory. Both buses consist of a 32-bit address bus, a 64-bit write data bus, and a 64-bit read data bus. This corresponds to a sustained bandwidth of 2.8 Gbytes/s per bus. The MTE is a multithreaded DMA engine with four MTCs. An MTC moves a block of data from a source address to a destination address. The MTE is a modified version of the DSE with four program counters (instead of one) as well as 128 registers and 2K of instruction memory. MTCs also have special functional units for BitBLT, Reed Solomon, and CRC operations.

- **Synchronization** : Each Quad has 32 globally accessible semaphore registers that are allocated either statically or dynamically. The semaphore registers associated with a PE, when set, can also generate interrupts to the PE.

1.2.3 Software Architecture and Tools

The 3SoC chip can be programmed using standard ANSI C or a C-like assembly language ("CLASM") or a combination thereof. The chip is supplied with GNU-based optimizing C-compilers, assemblers, linkers, debuggers, a functional and performance accurate simulator, and advanced code profilers and performance analysis tools. Please refer to 3SoC programmer's guide [4].

1.3 THE OPENMP COMPILER/TRANSLATOR

There are a number of OpenMP implementations for C and FORTRAN on SMP machines today. One of the approaches is to translate a C program with OpenMP directives to a C program with Pthreads [5]. Design of portable OpenMP compiler and translations of OpenMP directives have been studied [7, 8, 9, 10]. Our OpenMP prototype compiler consists of three phases as described in the following subsections.

1.3.1 Data Distribution

In OpenMP, there are several clauses to define data privatization. Two major groups of variables exist: shared and private data. Private data consists of variables that are accessible by a single thread or processor that doesn't need communication, such as variables defined in "PRIVATE" and "THREADPRIVATE" clause. Some private data needs initialization or combination before or after parallel constructs, like "FIRSTPRIVATE" and "REDUCTION". Access to these data should be synchronized among different processors.

1.3.2 Computation Division

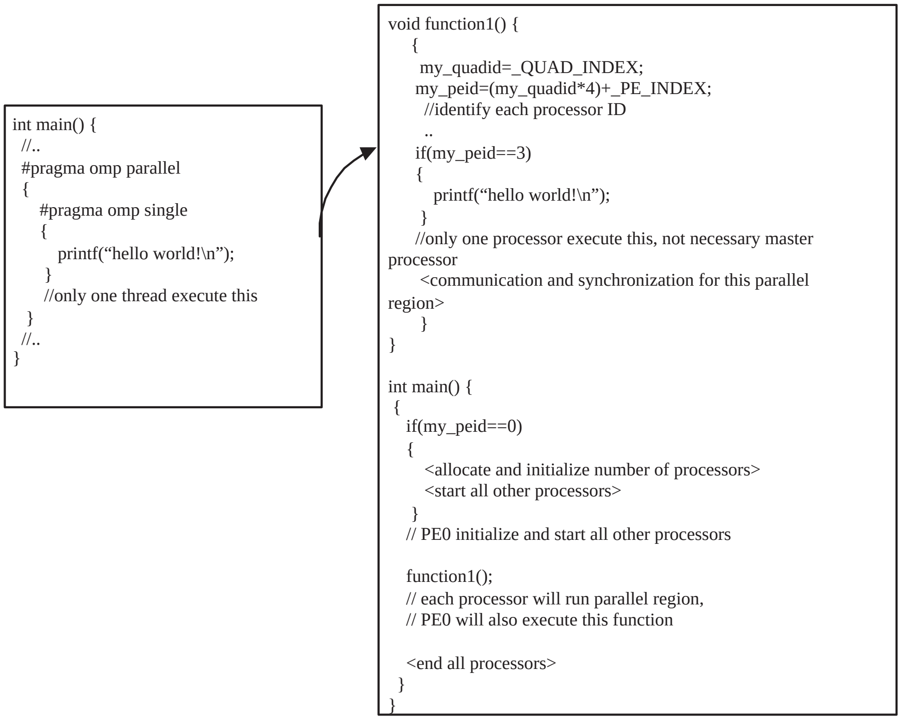
The computation needs to be split among different processors. The only way to represent parallelism in OpenMP is by means of PARALLEL directive as shown below:

```
#pragma omp parallel
{
    /* code to be executed in parallel */
}
```

In the 3SoC architecture, a number of processors can be viewed as a number of "threads" compared to normal shared memory architectures. Each processor or "thread" has its own private memory stack. At the same time, each processor is accessing the same blocks of shared local memory within the Quad or SDRAM outside Quad. In a typical 3SoC program, PE0 will initiate and start several other processors like PEs or DSEs, so that PE0 acts as the "master" thread and all other processors act as "child" threads. Then PE0 will transfer parameters and allocate data among different processors. It will also load the MTE firmware and enable all MTCs. Through data allocation PE0 tells each processor to execute specific regions in parallel. PE0 will also execute the region itself as the master thread of the team. At the end of a parallel region, PE0 will wait for all other processors to finish and collect required data from each processor, similar to a "master" thread.

The common translation method for parallel regions uses a micro-tasking scheme. Execution of the program starts with the master thread, which during initialization creates a number of spinner threads that sleep until they are needed. The actual task is defined in other threads that are waiting to be called by the spinner. When a parallel construct is encountered, the master thread wakes up the spinner and informs it the parallel code section to be executed and the environment to be setup for this execution. The spinner then calls the task thread to switch to a specific code section and execute.

For a chip multiprocessor environment, each "thread" unit is one processor. The number of "threads" is the actual processor number instead of a team of virtual threads, which can be created at the discretion of the user in a normal shared memory model. It is not practical to create two threads - one for spinning and another for actual execution. Moreover, each processor has its own processing power and doesn't wait for resources from other processors. In our approach, we simply assign each parallel region in the program with a unique identifying function. The code inside the parallel region is moved from its original place and replaced by a function statement,



```
int main() {
  //..
  #pragma omp parallel
  {
     #pragma omp single
     {
        printf(“hello world!\n”);
      }
      //only one thread execute this
   }
  //..
}
```

```
void function1() {
     {
      my_quadid=_QUAD_INDEX;
     my_peid=(my_quadid*4)+_PE_INDEX;
       //identify each processor ID
       ..
     if(my_peid==3)
     {
         printf(“hello world!\n”);
      }
     //only one processor execute this, not necessary master
processor
      <communication and synchronization for this parallel
region>
      }
}

int main() {
 {
   if(my_peid==0)
   {
       <allocate and initialize number of processors>
       <start all other processors>
    }
   // PE0 initialize and start all other processors

   function1();
   // each processor will run parallel region,
   // PE0 will also execute this function

   <end all processors>
  }
}
```

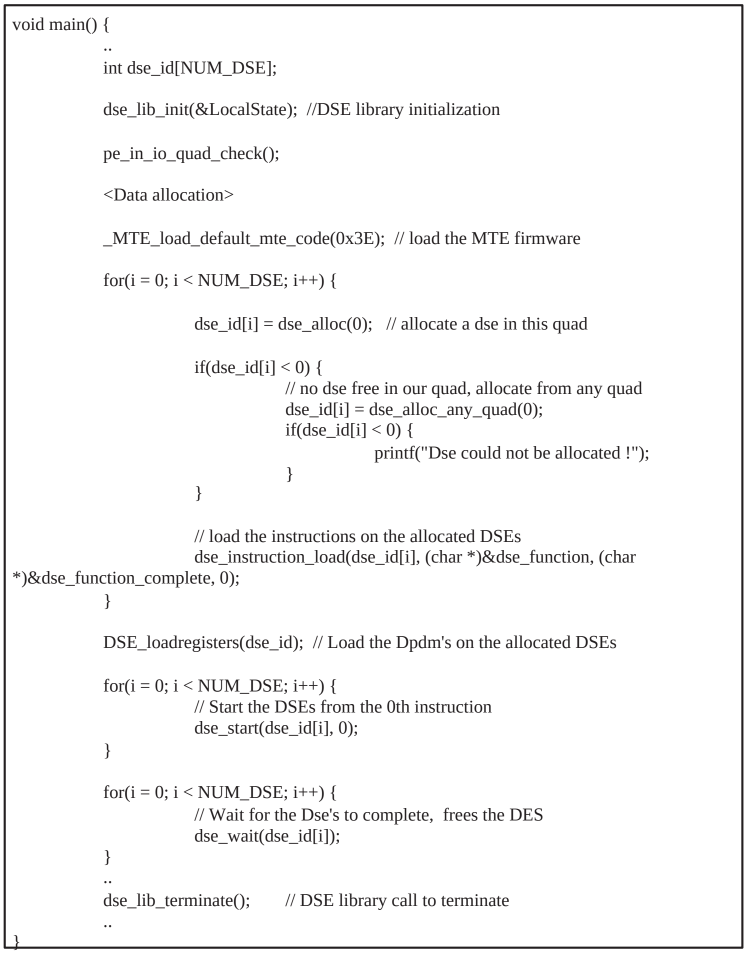
Fig. 1.3 Translation of an OpenMP program (left) to a 3SoC parallel region (right).

where its associated region calls this function and processors with correct IDs execute selected statements in parallel (See Fig. 1.3).

1.3.3 Communication Generation

In OpenMP specifications, several communications and synchronizations need to be guaranteed and inserted into the parallel regions at certain points. For example, only one processor is allowed access to the global "REDUCTION" variable at the end of the parallel construct at a time before an implicit barrier. Hardware synchronization features like semaphores in 3SoC are the most important features that distinguish normal multiprocessor chips from "parallel" chips. On 3SoC platform, the semaphore library (Semlib) has procedures for allocating global semaphores and Quad semaphores and for locking and unlocking. Reading a semaphore register, which also sets the register, is an atomic operation that cannot be interrupted. Sample barrier code is shown below:

```
semaphore_lock(Sem1.p);
  done_pe++;             //global shared variable
semaphore_unlock(Sem1.p);
while(done_pe<(PES)); //PES is total
```



```
void main() {
        ..
        int dse_id[NUM_DSE];

        dse_lib_init(&LocalState);  //DSE library initialization

        pe_in_io_quad_check();

        <Data allocation>

        _MTE_load_default_mte_code(0x3E);  // load the MTE firmware

        for(i = 0; i < NUM_DSE; i++) {

                dse_id[i] = dse_alloc(0);   // allocate a dse in this quad

                if(dse_id[i] < 0) {
                        // no dse free in our quad, allocate from any quad
                        dse_id[i] = dse_alloc_any_quad(0);
                        if(dse_id[i] < 0) {
                                printf("Dse could not be allocated !");
                        }
                }

                // load the instructions on the allocated DSEs
                dse_instruction_load(dse_id[i], (char *)&dse_function, (char
*)&dse_function_complete, 0);
        }

        DSE_loadregisters(dse_id);  // Load the Dpdm's on the allocated DSEs

        for(i = 0; i < NUM_DSE; i++) {
                // Start the DSEs from the 0th instruction
                dse_start(dse_id[i], 0);
        }

        for(i = 0; i < NUM_DSE; i++) {
                // Wait for the Dse's to complete,  frees the DES
                dse_wait(dse_id[i]);
        }
        ..
        dse_lib_terminate();       // DSE library call to terminate
        ..
}
```

Fig. 1.4 Sample code for controlling DSEs.

```
number of PEs
  _pe_delay(1);
```

1.4 EXTENSIONS TO OPENMP FOR DSES

Programming for PEs is similar to conventional parallel programming. The programs start with one PE (PE0) that is responsible for the environment setup and initialization of all other PEs. Afterwards PEs are involved in execution of selected statements within each parallel region by its associated processor ID. PEs are the primary processing units. Our implementation of OpenMP compiler could accept standard C programs with OpenMP directives and successfully convert it to parallel programs for PEs. The heterogeneity is based on DSE processors.

1.4.1 Controlling the DSEs

The controlling PE for a given DSE has to load the DSE code into the DSE instruction memory. Thereafter, the PE initializes the DSE DPDMs with the desired variables and starts the DSE. The PE then either waits for the DSE to finish, by polling, or can continue its work and get interrupted when the DSE finishes its task. Several DSE library calls are invoked. Sample program is shown in Fig. 1.4.

First, the PE initializes the DSE library calls via dse_lib_init(&LocalState). Then the PE does Quad I/O check and data allocation such as assigning initial value for the matrix. In the next for-loop, the PE allocates a number of DSEs and loads the DSE code into the DSE instruction memory by dse_instruction_load(). This is done by allocating within one Quad first, dse_id[i]= dse_alloc(0), if failed, it will load from other Quads. Afterwards, the PE loads the DPDM's onto the allocated DSEs, DSE_loadregisters(dse_id). After all initializations are done, the PE starts all DSEs and tells DSEs to execute from the 0th instruction, via the function call dse_start(dse_id[i], 0). The PE then waits for the DSEs to finish and automatically releases all DSEs, by dse_wait(dse_id[i]). When all tasks finish, the DSE terminate library call dse_lib_terminate() is invoked.

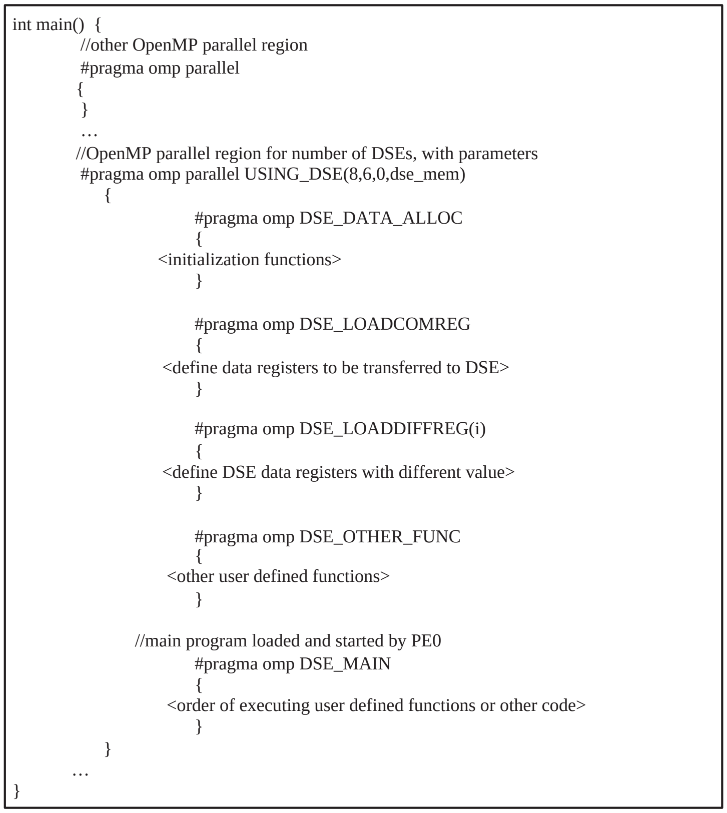
1.4.2 Extensions for DSEs

The main parallel region is defined as *#pragma omp parallel USING_DSE(parameters)*. When the OpenMP compiler encounters this parallel region, it will switch to the corresponding DSE portion. The four parameters declared here are: number of DSEs, number of Registers, starting DPDM number, and data register array, such as (8, 6, 0, dse_mem). For OpenMP compiler, the code generation is guided by the parameters defined in parallel USING_DSE construct. The compiler will generate environment setup like dse_lib_init, dse_alloc(0), DSE startup and wait call dse_start(), dse_wait(), and termination library call dse_lib_terminate(). So users are not required to do any explicit DSE controls, like startup DSE dse_start(). Fig. 1.5.

The benefit of using extensions is that it helps to abstract high-level parallel programs, and allows the compiler to insert initialization code and data environment setup, if required. This hides DSE implementation details from the programmer and greatly improves the code efficiency for parallel applications.

1.5 OPTIMIZATION FOR OPENMP

In a chip multiprocessor environment, several unique hardware features are specially designed to streamline the data transfer, memory allocations, etc. Such features are important to improve the performance for parallel programming on CMP. Some optimizations use a thread library for OpenMP which provides runtime optimization of parallel regions [11]. In this section, we present some optimization techniques that can be deployed to fully utilize advanced features of 3SoC, thus improving the performance for OpenMP.



```
int main()  {
          //other OpenMP parallel region
          #pragma omp parallel
         {
          }
          …
         //OpenMP parallel region for number of DSEs, with parameters
          #pragma omp parallel USING_DSE(8,6,0,dse_mem)
              {
                            #pragma omp DSE_DATA_ALLOC
                            {
                      <initialization functions>
                            }

                            #pragma omp DSE_LOADCOMREG
                            {
                       <define data registers to be transferred to DSE>
                            }

                            #pragma omp DSE_LOADDIFFREG(i)
                            {
                       <define DSE data registers with different value>
                            }

                            #pragma omp DSE_OTHER_FUNC
                            {
                       <other user defined functions>
                            }

                   //main program loaded and started by PE0
                            #pragma omp DSE_MAIN
                            {
                       <order of executing user defined functions or other code>
                            }
              }
         …
}
```

Fig. 1.5 Extensions to OpenMP for DSEs.

1.5.1 Using MTE Transfer Engine

Memory allocation is critical to the performance of parallel programs on SOCs. Given the availability of local memory, programs will achieve better performance in local memory than in SDRAM. On-chip memory is of limited size for SOCs or other equivalent DSP processors. Data locality is not guaranteed. One approach is to allocate data in DRAM first, then move data from DRAM to local memory at run-time. Thus, all the computation is done in on-chip memory instead of the slow SDRAM. In 3SoC, developer can invoke one PE to move data between the local memory and DRAM at run-time.

3SoC also provides a better solution for data transfer using MTE transfer engine (detailed in Sec 1.2.2). Note that the MTE processor runs in parallel with all other processors. It transfers data between local data memory and SDRAM in the background. We use extensions to OpenMP to incorporate MTE transfer engine. The OpenMP directives are:

```
#pragma omp MTE_INIT(buffer size, data structure, data slice)
```

```
#pragma omp MTE_MOVE(count, direction)
```

MTE_INIT initializes a local buffer for data structure with specified buffer size. MTE_MOVE will perform actual data movement by MTE engine. Data size equaling count*slice will be moved with respect to the direction (from local->DRAM or DRAM->local). Within a parallel region, a developer can control data movement between local memory and SDRAM before or after the computation. The MTE firmware needs to be loaded and initiated by PE0 at the beginning of the program. A number of MTE library calls will be generated and inserted by the compiler automatically. The results show significant performance speedup using the MTE to do data transfer, especially when the size of target data structure is large. Performance evaluation of using the MTE versus using the PE to do data transfer is given in Section 1.7.

1.5.2 Double Buffer and Data Pre-fetching

Data pre-fetching is a popular technique to improve the memory access latencies. Besides using the MTE to do data transfer in 3SoC, we can also apply a data pre-fetching approach through Double Buffering.

For non-Double-Buffering, as discussed in section 1.5.1, we assume data is allocated in SDRAM first. Before the PE starts to perform computations, it invokes the MTE engine to populate or move the data from DRAM to local memory. When the MTE is done, it will interrupt the PE informing it that data is ready and computation can be started. The interrupts used are semaphore interrupts. The PE locks a semaphore before calling on the MTE to move data. Once the MTE is done, it unlocks the semaphore thus causing an interrupt. To reduce the memory access latencies, double buffering is used to improve the performance. Instead of using one buffer in the previous example, it uses two local buffers which work in round-robin manner, each time one buffer is being computed, data in another buffer is being transferred, and vice versa.

Fig. 1.6 shows how to perform matrix multiplication using double buffering. We are multiplying matrices A and B, and the result is kept in matrix C. Matrix B is in the local memory, while matrices A and C are both in DRAM. However, instead of one local buffer per matrix, we allocate two buffers in the local memory for both matrices A and C. The PE calls the MTE to populate the first local buffer of matrix A. The PE then calls the MTE to populate the second local buffer of matrix A, while the MTE is moving data, the PE starts to perform computations, storing the result in the first local buffer of matrix C. Sometime during the computations, the PE will be interrupted by the MTE. When the PE finishes the first round of computation, it can start on the second local buffer of matrix A, and store the result in the second local buffer of matrix C. As a result, at any given time, while the PE is performing computations, the MTE will be moving data from the DRAM into a local buffer of matrix A and also will be moving the completed results from a local buffer of matrix C into the DRAM.

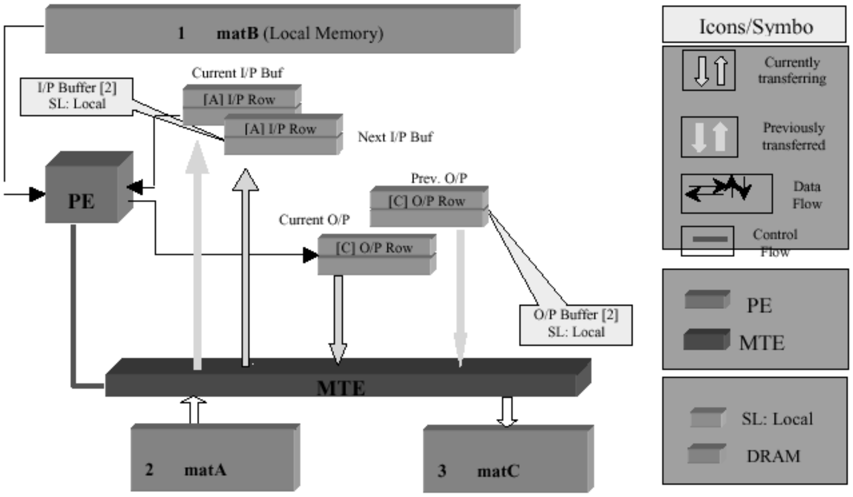


Fig. 1.6 Extensions to OpenMP for DSEs.

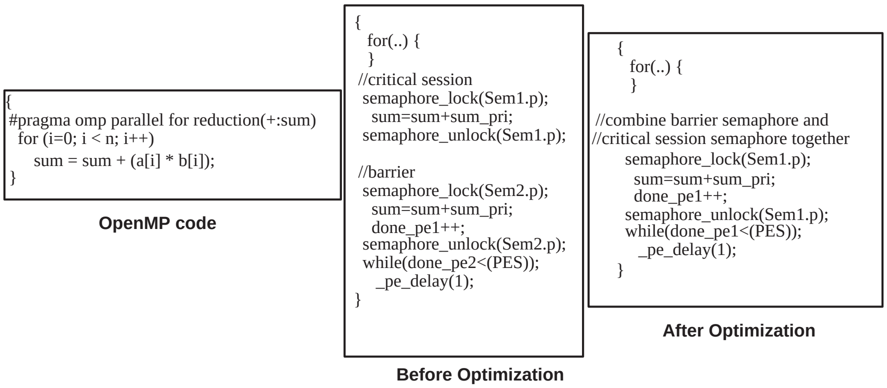
To implement Double Buffering to improve the performance for OpenMP, we provide extensions to OpenMP. Users are required to perform explicit control of data movement between local memory and SDRAM. The directives are:

```
#pragma omp DB_INIT(buffer1 size, buffer2 size, data structure1,
data structure2, data slice1, data slice2)
#pragma omp
DB_MOVE(buffer ID1, direction1, buffer ID2, direction2)
```

DB_INIT initializes two buffers for each data structure with specified size, totally four buffers. DB_MOVE at certain point controls the actual data movement between SDRAM and local memory. Each time DB_MOVE will move one slice for both data structure1 and structure2, with specified direction (from local->DRAM or DRAM->local) and buffer ID(1 or 2) for each data structure. Concurrently, PE will do computation against another buffer of each structure. The OpenMP Compiler automatically sets up the environment, initializes the MTE, allocates necessary buffers and inserts the required library calls. With the help of these extensions, users can write OpenMP parallel programs which control data movement dynamically at run-time.

1.5.3 Data Privatization and Others

OpenMP provides few features for managing data locality. The method provided for enforcing locality in OpenMP is to use the PRIVATE or THREADPRIVATE clause. However, systematically applied privatization requires good programming practices. Some researchers have proposed several approaches to provide optimization with modest programming effort, including the removal of barriers that separate two



```
{
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
      sum = sum + (a[i] * b[i]);
}
```

```
{
   for(..) {
   }
 //critical session
  semaphore_lock(Sem1.p);
    sum=sum+sum_pri;
  semaphore_unlock(Sem1.p);

 //barrier
  semaphore_lock(Sem2.p);
    sum=sum+sum_pri;
    done_pe1++;
  semaphore_unlock(Sem2.p);
  while(done_pe2<(PES));
     _pe_delay(1);
}
```

```
   {
     for(..) {
     }

 //combine barrier semaphore and
//critical session semaphore together
      semaphore_lock(Sem1.p);
        sum=sum+sum_pri;
        done_pe1++;
      semaphore_unlock(Sem1.p);
      while(done_pe1<(PES));
         _pe_delay(1);
   }
```

Fig. 1.7 Optimization (Semaphore Elimination).

consecutive parallel loops [12], improving cache reuse by means of privatization and other improvement [13, 14].

In order to improve the performance of OpenMP on 3SoC, we apply those optimization techniques. For the time being, not all techniques discussed here are available in our first version compiler.

For barrier elimination, it may be possible to remove the barrier separating two consecutive loops within a parallel region. Barriers require a lot of communication and synchronization such that this optimization can greatly improve the performance. For data privatization, shared data with read-only accesses in certain program sections can be made "PRIVATE" and treated as "FIRSTPRIVATE" which has copy-in value at the beginning of parallel regions. For the 3SoC architecture, all synchronization is carried out by means of hardware semaphores. It is helpful to combine these semaphores together when encountered with a consecutive barrier and critical section, thus reducing the overall synchronization. For example, at the end of the parallel region, the "REDUCTION" variable needs to be synchronized and modified by each thread to reflect the changes, which can be combined with an implicit barrier at the end of parallel region, as illustrated in Fig. 1.7.

1.6 IMPLEMENTATION

In this section, we discuss our implementation of the OpenMP compiler/translator for 3SoC. However, this chapter is not focused on implementation details. To implement an OpenMP compiler for 3SOC, there are four major steps [6].

- **Parallel regions** : Each parallel region in the OpenMP program will be assigned a unique identifying function number. The code inside the parallel region is moved from its original place into a new function context. The parallel construct code will be replaced by code of PE0's allocating multiple PEs or DSEs, setting up environment, starting all processors, assigning workload to each processor, and waiting for all other processors to finish.

- **Data range** : Through analysis of all the data attributes in the OpenMP data environment clause, i.e., "SHARED", "PRIVATE", "FIRSTPRIVATE", "THREADPRIVATE", "REDUCTION", compiler determines the data range for separate functions and assign memory allocation like "_SL", or "_SD" in 3SOC. Related global variable replication such as "REDUCTION" is also declared and implemented.
- **Work sharing constructs** : These are the most important constructs for OpenMP, referred as for-loop directive, sections directive and single directive. Based on the number of processors declared at the beginning of the 3SOC program, each processor will be assigned its portion of work distinguished by processor ID. During run-time, each processor will execute its own slice of work within designated functions. For example, for the "sections" construct, each sub-section defined in *#pragma omp section* will be assigned a distinct processor ID, and run in parallel by different processors.
- **Synchronization** : There are a number of explicit or implicit synchronization points for OpenMP constructs, i.e., critical, or parallel construct. Correspondingly, these constructs are treated by allocating a number of hardware semaphores in 3SOC. Allocation is achieved statically or dynamically.

Our current version of OpenMP compiler can take standard OpenMP programs. Provided with extensions to OpenMP, users can also write OpenMP code to utilize advanced chip multiprocessor features, like different processors, MTE or Double Buffering on 3SoC.

1.7 PERFORMANCE EVALUATION

Our performance evaluation is based on 3SoC architecture; the execution environment is the 3SoC cycle accurate simulator, Inspector (version 3.2.042) and the 3SoC processor. Although we have verified the programs on the real hardware, we present results on the simulator as it provides detailed profiling information.

To evaluate our OpenMP compiler for 3SoC, we take parallel applications written in OpenMP and compare the performance on multiple processors under different optimization techniques. The first parallel application is Matrix Multiplication. By applying different optimizations at compilation, we compare the performance of parallel application among: no optimization, with data locality (matrices in local memory), using the MTE for data transfer, using the PE for data transfer and double buffering separately. The second application is LU decomposition that follows the same approach. We also show the compiler overhead by comparing the result with hand-written code in 3SoC.

Fig. 1.8 shows the results of matrix multiplication using multiple PEs. The speedup is against sequential code running on single processor (one PE). Fig. 1.9 is the result for LU decomposition using multiple PEs against one PE. We use four

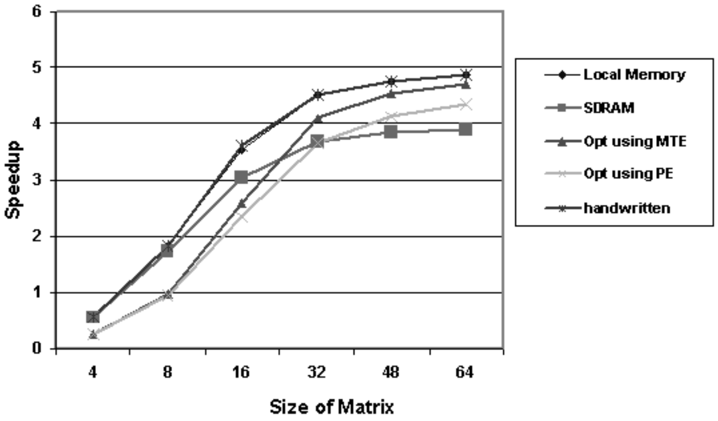


Fig. 1.8 Matrix Multiplication using 4 PEs.

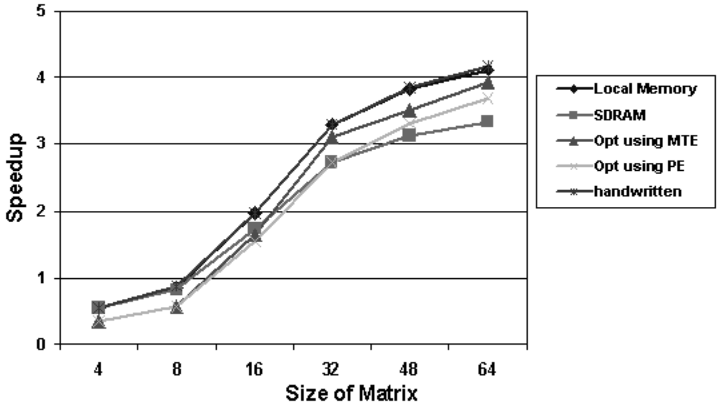


Fig. 1.9 LU Decomposition using 4 PEs.

PEs within one Quad for both cases. By analysis of both charts, we conclude the following:

- **Local memory vs SDRAM** : As expected, memory access latencies have affected the performance significantly. When the size of the data structure (matrix size) increases, speedup by allocation of data in local memory is obvious. For 64*64 matrix LU decomposition, the speedup is 4.12 in local memory vs 3.33 in SDRAM.
- **Using the MTE vs SDRAM** : As discussed in Section 5, we can deploy the MTE data transfer engine to move data from SDRAM to local memory at run-time, or we can leave the data in SDRAM only and never transferred to local during execution. Due to the limited size of the local memory it's not practical to put all data within the local memory. For small size matrices below

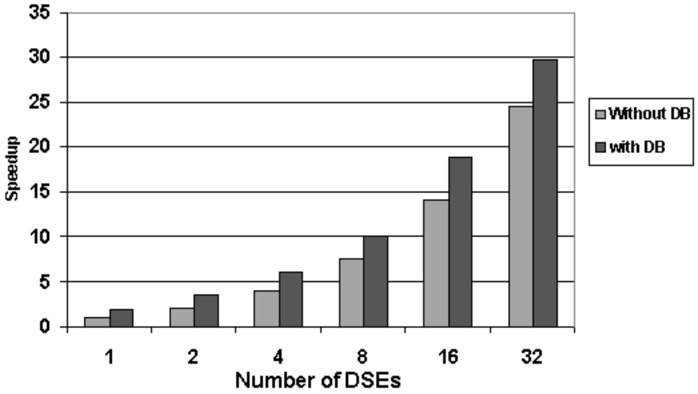


Fig. 1.10 Matrix Multiplication using DSEs.

32*32, the MTE transfer has no benefit, in fact, it downgrades the performance in both examples. The reason is that the MTE environment setup and library calls need extra cycles. For larger-size matrices, it shows speedup compared to data in SDRAM only. For 64*64 matrix multiplication, the speedup is 4.7 vs 3.9. Actually 64*64 using MTE engine is only a 3.2% degrade compared to storing data entirely in the local memory. Therefore, moving data using the MTE will greatly improve performance for large data.

- **Using the MTE vs using the PE** : We observed scalable speedup by using the MTE over the PE to move data. The extra cycles used in MTE movement do not grow much as the matrix size increases. For large data set movements, the MTE will achieve greater performance over the PE.
- **Using compiler generated vs hand-written code** : The overhead of using the OpenMP compiler is addressed here. Since the compiler uses a fixed allocation to distribute computation, combined with extra code added to the program, it is not as good as manual parallel programming. In addition, some algorithms in parallel programming cannot be represented in OpenMP. The overhead for OpenMP compiler is application dependent. Here we only compare the overhead of the same algorithm deployed by both the OpenMP compiler and handwritten code. It shows overhead is within 5% for both examples.

Fig. 1.10 shows the result of matrix multiplication using multiple DSEs. Double Buffering techniques are used here. The matrix size is 128*128.

- **Scalable speedup by using a number of DSEs** : 4 DSEs achieve 3.9 speedup over 1 DSE for the same program without double buffering, and 32 DSEs obtain 24.5 speedup over 1 DSE. It shows that 3SoC architecture is suitable for large intensive computation on multiple processors within one chip and performance is scalable.
- **Double Buffering** : Double buffering shows great performance improvement, especially for smaller numbers of DSEs. For 1 DSE, the speedup is 1.8 by

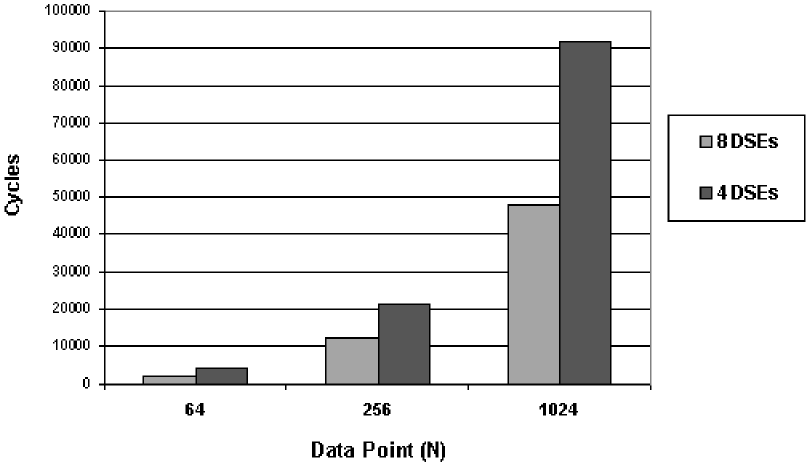


Fig. 1.11 Parallelized FFT using DSEs.

using DB over 1 DSE without DB, almost equivalent to using two DSEs. We expect the speedups with larger number of DSEs to be in the same range with larger matrices.

In Fig. 1.11, we implemented parallelized FFT using multiple DSEs. For most applications computation time plays an important role in the use of FFT algorithm. The computation time can be reduced using parallelism in FFT, in 3SoC, employing multiple DSEs. Fig. 1.11 shows the scalable scheme of FFT using different number of DSEs. From the computation cycles taken, the time for computation of 1024 complex points using 8 DSEs is approximately 240 microseconds (with the current 3SoC clock speed of 200Mhz), which is comparable to other DSP processors. For 64 fixed size data points, using 8 DSEs achieves 1.95 speedup over 4 DSEs. It is clear from Fig. 1.11 that FFT implementation in OpenMP is scalable.

1.8 CONCLUSIONS

In this paper, we propose an OpenMP compiler for chip multiprocessors (3SoC as an example), especially targeting at extending OpenMP directives to cope with the heterogeneity of CMPs. In view of this emerging parallel architecture, advanced architecture feature is important. By extending OpenMP for CMPs, we provide several optimization techniques. The OpenMP compiler hides the implementation details from the programmer, thus improving the overall code efficiency and ease of parallel programming on CMPs.

REFERENCES

1. OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 2.0, *http://www.openmp.org*, March 2002.

2. R.Chandra, L.Dagum, D.kohr, D.Maydan, J.McDonald, and R.Menon. Parallel Programming in OpenMP. Morgan Kaufmann Publishers, 2001.

3. 3SoC Documentation–3SoC 2003 Hardware Architecture, Cradle Technologies, Inc. Mar 2003.

4. 3SoC Programmer's Guide, Cradle Technologies, Inc., *http://www.cradle.com*, Mar 2002.

5. Christian Brunschen, and Mats Brorsson, OdinMP/CCp - A portable implementation of OpenMP for C, In *uropean Workshop on OpenMP*, Sepetember 1999.

6. Feng Liu, Vipin Chaudhary, A practical OpenMP compiler for System on Chips, *Workshop on OpenMP Applications and Tools*, pages 54-68, June 2003.

7. Dan Quinlan, Markus Schordan, Qing Yi and Bronis R. De. A C++ Infrastructure for automatic Introduction and Translation of OpenMP Directives, *workshop on OpenMP Applications and Tools*, Jun 2003.

8. Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP compiler for an SMP cluster. In *European Workshop on OpenMP*, September 1999.

9. Seung Jai Min, Seon Wook Kim, Michael Voss, Sang Ik Lee, and Rudolf Eighmann. Portable compilers for OpenMP. In *Workshop on OpenMP Applications and Tools*, 2001.

10. Xinmin Tian, Aart Bik, Milind Girkar Paul Grey, Hideki Saito, and Ernesto Su. Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1):36-46, 2002.

11. Mihai Burcea and Michael J. Voss A Runtime Optimization System for OpenMP. *Workshop on OpenMP Applications and Tools*, June 2003.

12. C. Tseng, Compiler optimization for eliminating barrier synchronization, *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

13. S. Satoh, K. Kusano, and M. Sato. Compiler Optimization Techniques for OpenMP Programs, *2nd European Workshop on OpenMP*, pp 14-15, 2000.

14. Arvind Krishnamurthy and Katherine A. Yelick. Optimizing parallel programs with explicit synchronization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196-204, 1995.