C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-1:Multi-Core)

Lecture Topic:

Multi-Core Processors :

Tuning & Performance /Compilers Part-II

Venue : CMSD, UoHYD Date : October 15-18, 2013

Tuning & Performance - PThreads Model

Lecture Outline

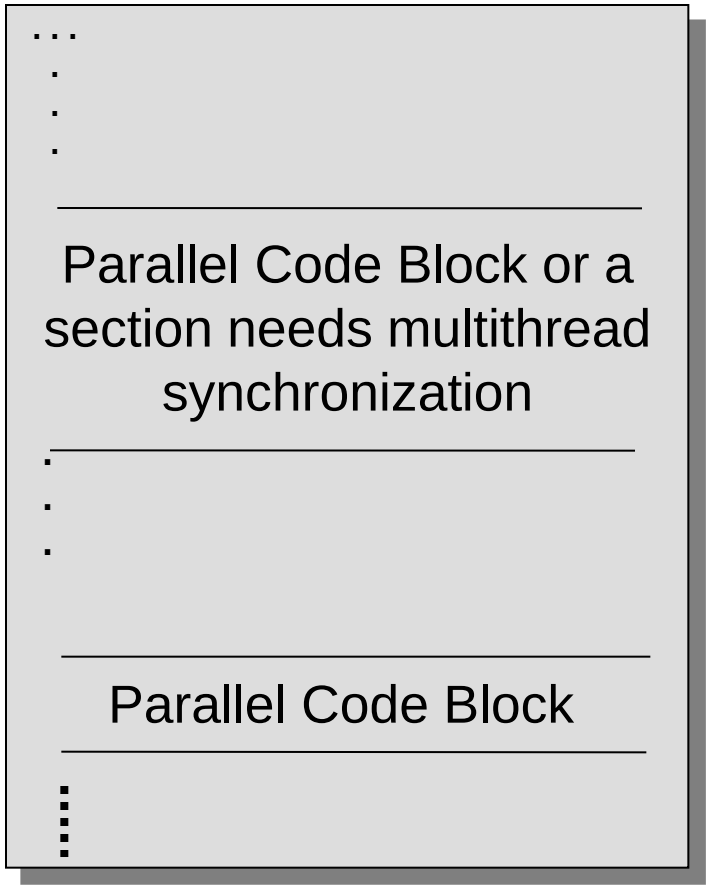
Following Topics will be discussed

- An overview Tuning & Performance on Multi Cores
- Understanding Code Restructuring on Single Core /Multi Core – Compiler Switches
- Multi Core Programming - Performance Issues
- Performance issues - Examples on Single /Multiple Cores

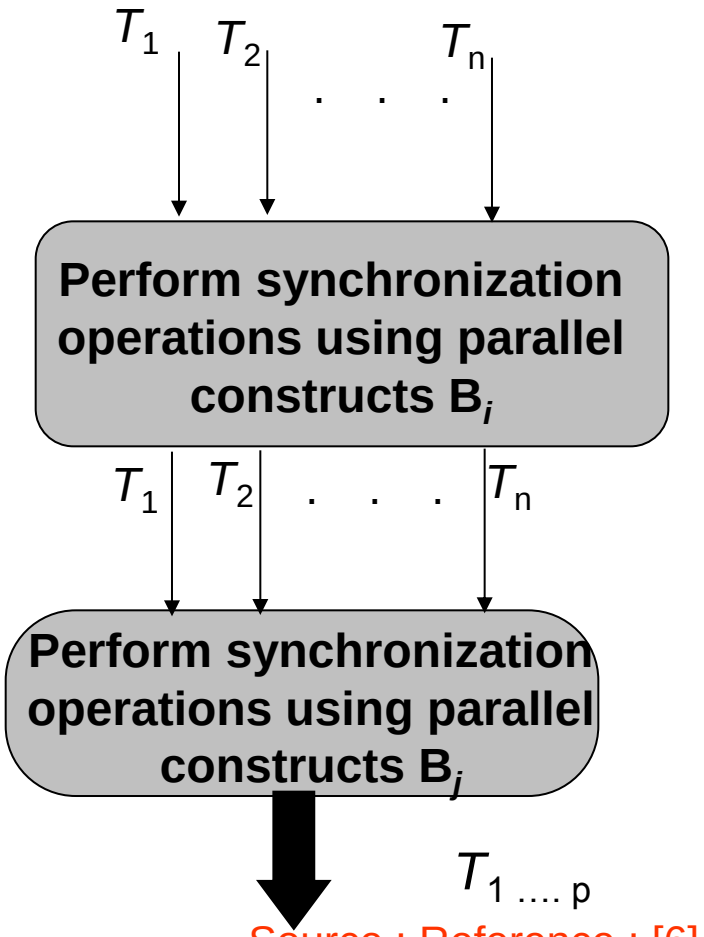
Source : Reference [4],[6], [7], [8], [32], 34]

Operational Flow of Threads for an Application

Implementation Source Code



Operational Flow of Threads



Source : Reference : [6]

Programming Multicore Processors

- Explicit Parallel Programming
  - Thread-based Programming Models.
  - Data Parallel Programming Models
  - Stream Programming Models
- Auotomatic Parallelization
  - Features of Most compliers for SMP systems, but currently see very little practical use
  - Polyhedral framework for dependencies and loop transformations – enabling composition of complex transformations over multiple statements.

Source : Reference [4],[6], [7], [8], [32], 34]

Using Your Compiler Effectively : Basic Compiler Tech.

Questions to be addressed :

- How Compiler optimizations can help user to get good performance?
- What you can do in your sequential program to get Single Core performance?
- What can you do in your parallel program to get good performance on given Multi Cores ?
- What advanced things can I do with the compiler to help get good performance as well as correct results on Multi Cores ?

Remarks:

- It is important to know whether the compiler is compiling the code optimally so that you can adjust the code, compiler options or something else on Single /Multi Cores

Improving Single Core Performance

- How much sustained performance one can achieve for given program on a Multi Cores ?
- It is programmer's job to take advantage as much as possible of the Cores hardware /software characteristics to boost the performance of the program on Multi Cores !
- Quite often, just a few simple changes to one's code improves performance by a factor of 2, 3 or better !
- Also, simply compiling with some of the optimization flags (-O3, -fast, ….) can improve the performance dramatically on Multi Cores

Single Core Performance: Compiler Optimization

- Performance on Multi Core has to do with the following :
  - Problem size and precision (Role of Compiler)
  - Execution time - Computational Issues (Role of Compilers)
  - Ease of Programming (Role of Compilers)
  - Number of runnable threads /System Supported Cores
- Questions to be addressed
  - How big a problem can I solve on Multi cores ?
  - How precise is the solution of my problem ?
  - How long will it take for the Multi Cores to run the program to completion ?

Multi Core : Performance oriented Prog.

❖**Two issues to be addressed**

- How well does the single-threaded version run ?
- How well can the work be divided up among multiple processors with the least amount of overhead ?
- Are we implemented well-designed algorithm ?
- Are we implemented well-tuned application ?

Multi Core : Programming Models

- Data Parallel models
  - Microsoft Research Accelerator
- Multi-threaded Models
  - OpenMP, MPI
  - Cilk
  - CUDA
- Streaming Models
  - Streamit
  - Cilk
  - Peakstream (Brook)

Source : Reference [4],[6], [7], [8]

Compiler Techniques : Detailed Code Profiling

- Find out the routines that are taking the most time.
- Compiling with `suitable option` will instrument a code so that the "`gprof`" command can produce a first-cut picture of where time is being spent.
- Use Multi Core Profilers & Thread Checkers to understand the behavior of code

Timing Code :

- The first step in optimizing code is to profile it.
- Identify hot spots.
- It is often wise to time these regions by hand using a real time clock.

Source : Reference [4],[6], [7], [8], [32], 34]

Parallel Programming – Compiler switches

Remarks

- In some cases, parallelizing a reduction loop can give different answers depending on the number of cores on which the loop is run.
- Compiler directives can usually over come artificial barriers to parallelization.
- Compiler directives can also over come legitimate barriers to parallelization, which introduces errors.
- The efficiency and effectiveness of automatic compiler parallelization on Multi Core Systems can be significantly improved by supplying the switches.

Source : Reference [4],[6], [7], [8]

Muti Core : Performance oriented Prog.

❖ Challenges in Threading Loop

- ➢Loop-carried Dependence
- ➢Data Race Conditions
- ➢Managing Shared and Private Data
- ➢Loop Scheduling and Portioning
- ➢Effective use of Reductions

❖ Minimizing Thread Overheating

- ➢Work-Sharing Sections

Source : Reference [4],[6], [7], [8]

Multi Core : Performance oriented Prog.

- Too Many Threads
- Data Races, Deadlocks, and Live Locks
- Heavily Contented Locks
- Thread Safe functions and Libraries
- Memory Issues
- Cache-related Issues

Multi Core : Performance oriented Prog.

- The Underlying performance of the single-threaded code
- The percentage of the program that is run in parallel and its scalability
- CPU utilization, effective data sharing, data locality and load balancing
- The amount of synchronization and communication among the threads
- Memory Conflicts caused by shared memory or falsely shared memory.

Source : Reference [4],[6], [7], [8]

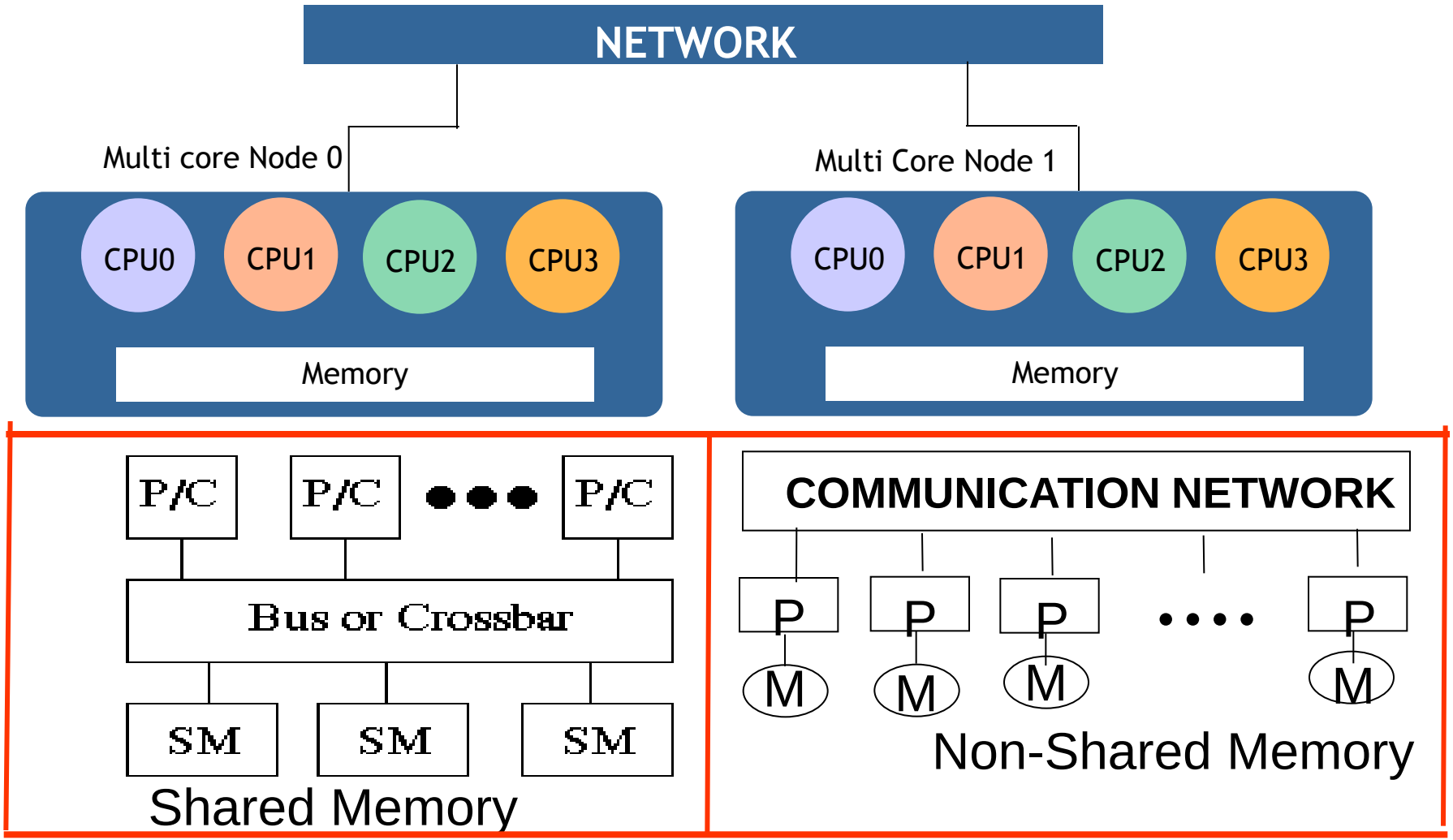
Multi Core : Performance oriented Prog.

- The overheads introduced to
  - Create
  - resume
  - Manage
  - Suspend
  - Destroy,
  - Wait
  - Synchronize

- Performance Limitations
  - Shared resources – Memory, write combining buffers, bus bandwidth, and CPU execution units

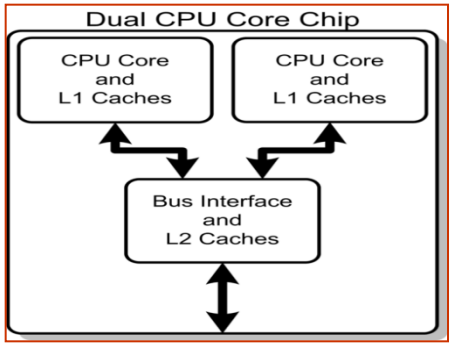
Source : Reference [4],[6], [7], [8]

General-Purpose Clusters /Multi Cores

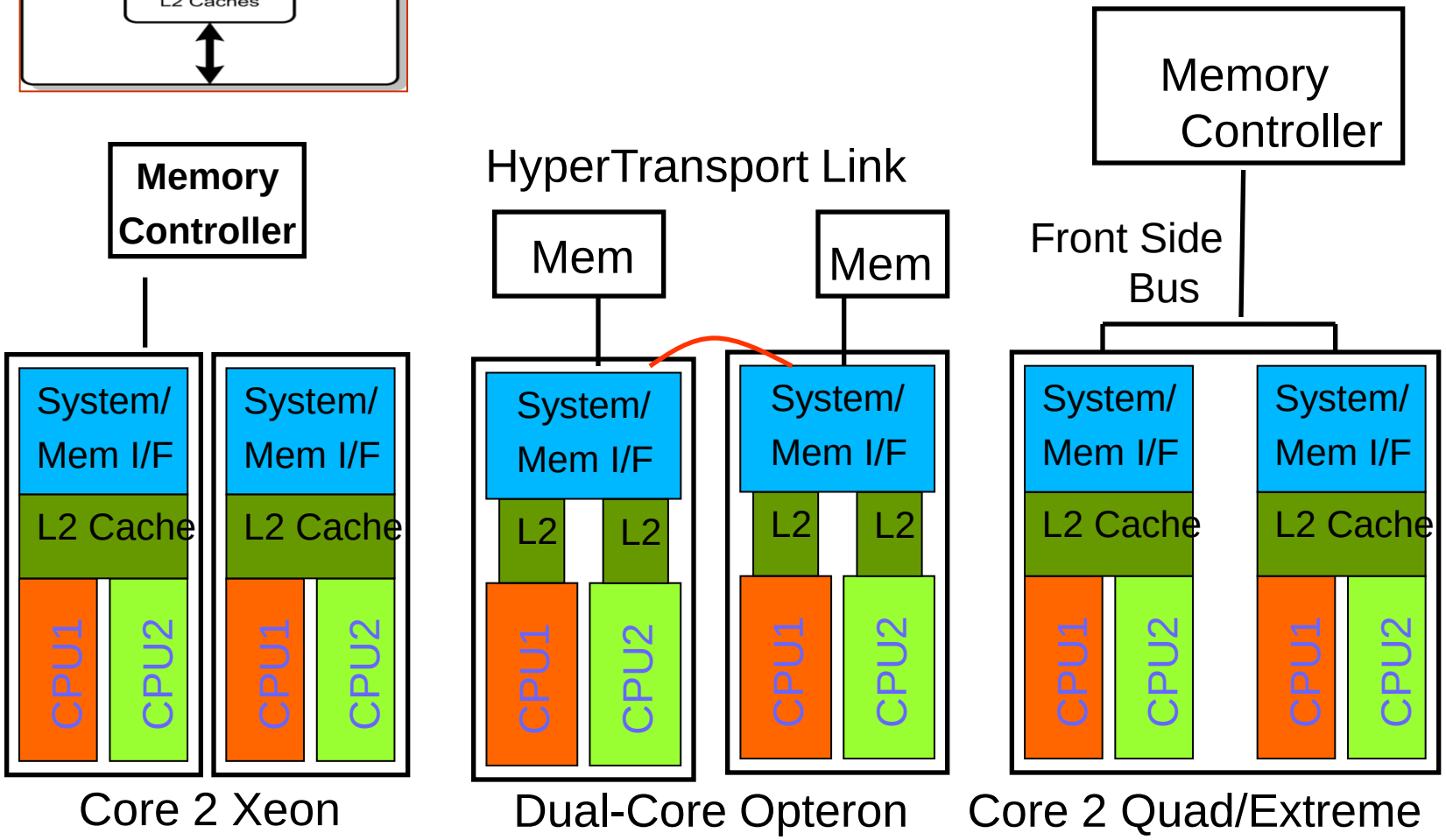


Source : http://www.intel.com; http://www.amd.com; Reference [4], [6]

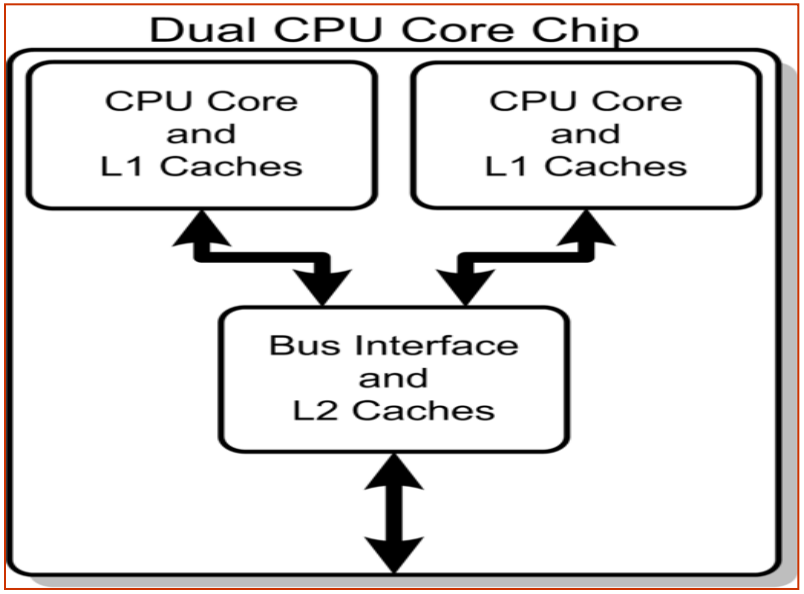
Multi Cores Processors



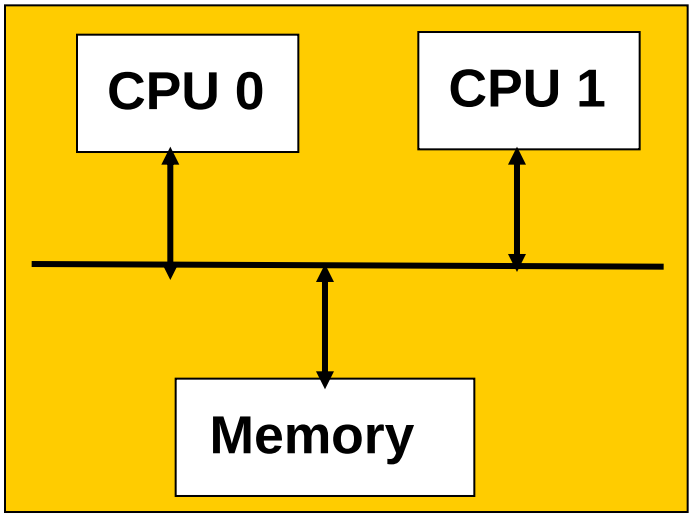
Source : http://www.intel.com; http://www.amd.com



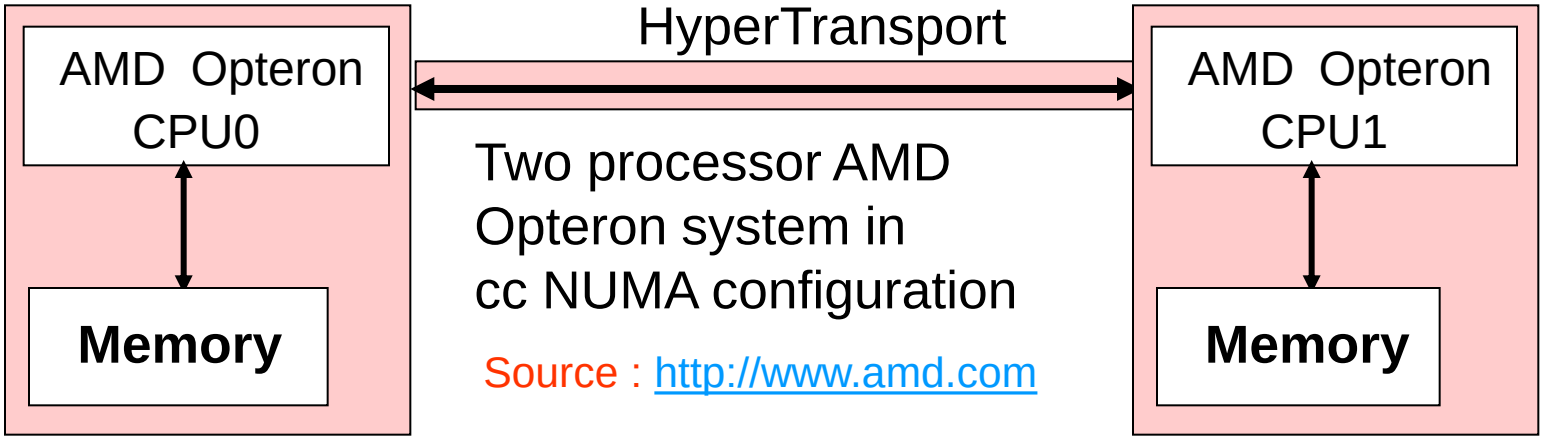
Multi Cores Processors



Two processor Dual Core



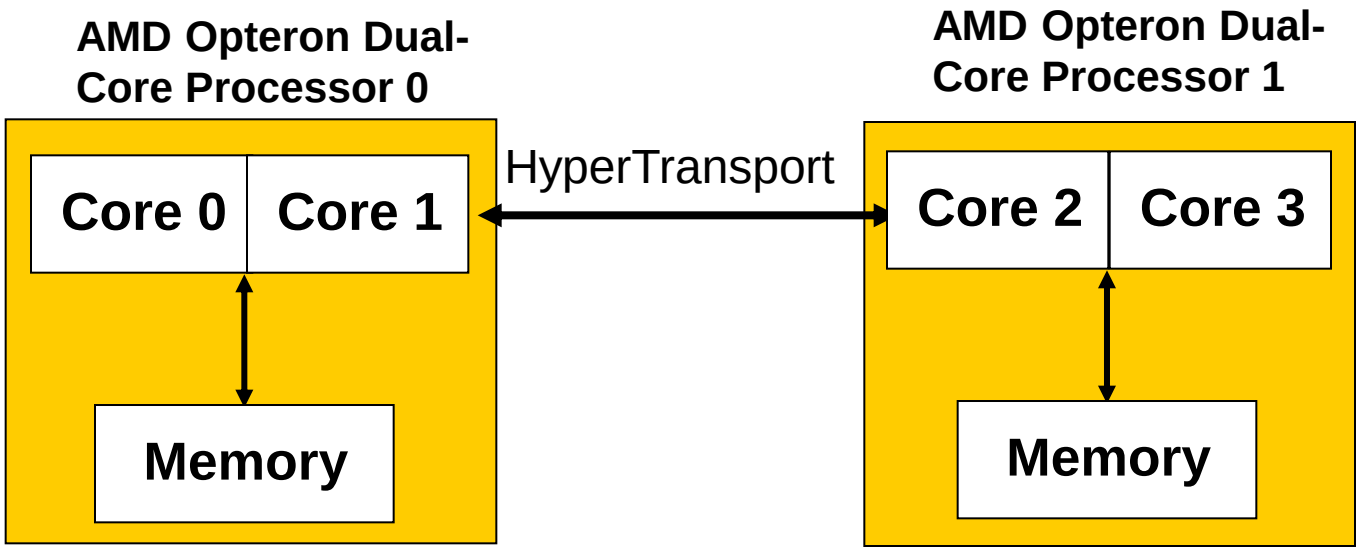
Simple SMP Block Diagram for a two processors



Two processor AMD Opteron system in cc NUMA configuration

Source : http://www.amd.com

Multi Cores Processors



Dual-Core AMD Opteron Processor configuration

- AMD : Cache-Coherent nonuniform memory access (ccNUMA)
  - Two or more processors are connected together on the same motherboard
  - In ccNUMA design, each processor has its own memory system.
  - The phrase 'Non Uniform Memory access' refers to the potential difference in latency

Source : http://www.amd.com

Gains from tuning categories

Tuning Category	***Typical Range of Gain***
Source range	***25-100%***
Compiler Flags	***5-20%***
Use of libraries	***25-200%***
Assembly coding / tweaking	***5-20%***
Manual prefetching	***5-30%***
TLB thrashing/cache	***20-100%***
Using vis.inlines/micro-vectorization	***100-200%***

Source : Reference [4],[6], [7], [8], [32], 34]

Loop Optimization Techniques

- Classical Optimization techniques – Compiler Does
- Memory Reference Optimization – Compiler does to some extent
- Loop Optimizations – Compiler does to some extent
- Loop Fission and Loop Fusion
  - Loop distribution
  - Loop Interchange
  - Loop Alignment
  - Loop Collapsing
  - Loop Unrolling
- The programmer should be cautious when Loop Optimizations are performed on Multi Core Processors.

Source : Reference [4],[6], [7], [8], [32], 34]

Source Code Optimizations

- Improve usage of data cache, TLB
- Use VIS instructions (templates) directly, via –xvis option
- Optimize data alignment (also: #pragma align,dalign)
- Prevent Register Window overflow
- Creating inline assembly templates for performance critical routines
- Loop Optimizations that compilers may miss:
  - Restructuring for pipelining and prefetching
  - Loop splitting/fission
  - Loop Peeling
  - Loop interchange
  - Loop unrolling and tiling
  - Pragma directed

Source : Reference [4],[6], [7], [8], [32], 34]

Loop Optimization Techniques

- Dependence Analysis
- Transformation Techniques
- Loop distribution
- Loop Alignment
- Node Splitting
- Strip Mining
- Loop Collapsing
- Loop Fission Loop Fusion
- Wave front method
- Loop Optimizations

More about Loop Optimizations

- Basic Loop Unrolling & Qualifying Candidates for Loop Unrolling
- Negatives of Loop Unrolling
- Outer and Inner Loop Unrolling
- Associative Transformations
- Loop Interchange

Source : Reference [4],[6], [7], [8], [32], 34]

Loop Unrolling Issues on Multi Cores

- Loop unrolling always adds some run time to the program.
- If you unroll a loop and see the performance dip little, you can assume that either:
  - The loop wasn't a good candidate for unrolling in the first place
    or
  - A secondary effort absorbed your performance increase.
- Other possible reasons
  - Unrolling by the wrong factor –Data Race Conditions
  - Register spitting
  - Instruction cache miss – False Sharing of Data
  - Other hardware delays -
  - Outer loop unrolling - Data Re-Use in the Caches of Multi Cores

Loop Distribution

- Loop distribution aims at distributing operations in such a way that the need for explicit synchronization is reduced.
- It changes the execution order but keeps the same statements.
- It can not eliminate dependencies but it can rearrange them and in particular, it can help in changing loop-carried into loop-independent dependencies.

Original

```
DO 10 I=1, N
   C(I)=A(I)+B(I)
   D(I)=A(I-1)+B(I)
10 CONTINUE
```

Modified

```
DO 10 I=1, N
   C(I)=A(I)+B(I) 10
CONTINUE
DO 20 I=1, N
   D(I)=C(I-1)x B(I)
20 CONTINUE
```

Loop Collapsing

- It attempts to create one (larger) loop out of two or more small ones.
- This may be profitable if the size of each of the two loops is too small for efficient vectorization, but the resulting single loop can be profitably vectorized.

Before

```
      REAL A(5,5) B(5,5)
     DO 10 J =1, 5
     DO 10 I=1, 5
      A(I,J) = B(I,J) + 2.0
10 CONTINUE
```

After

```
      REAL A(25) B(25)
     DO 10 JI =1, 25
      A(JI) = B(JI) +2.0
10 CONTINUE
```

- Loop collapsing is done with multi-dimensional arrays to avoid loop overheads

Source : Reference [4],[6], [7], [8], [32], 34]

Loop Collapsing

(Contd…)

- General Versions of this technique is useful for computing systems which support only a single (not nested) DOALL statement.

Before

```
DO 10 J =1, N
  DO 10 I =1, M
     A(I,J) = B(I,J) + 2.0
10 CONTINUE
```

After

```
DO 10 L = 1, NxM
        I  = (L-1)/M+1
         I  = MOD(L-1,M) +1)
    A(I,J) = B(I,J) + 2.0
10 CONTINUE
```

- Using this technique, the code may be transferred into a single loop, regardless of the size of M and N
- This may require some additional statement to restart the code properly.

Loop Collapsing

(Contd…)

- Loop collapsing is done with multi-dimensional arrays to avoid loop overheads
- Assume declaring **a[50][80][4]**

Un Collapsed Loop

```
for(I = 0; I <50;  i++)
for(j = 0;  j<80;  j++)
for(k = 0; k<4;  k++)
 a[i][j][k] = a[i][j][k] * b[i][j][k] + c[i][j][k];
```

Collapsed Loop

```
 for(i=0; i<50*80*4; i++)
a[0][0][k]  = a[0][0][k] * b[0][0][k] + c[0][0][k];
```

- **Warning :** This works only if the entire array space is accessed !

Loop Fission and Loop Fusion

Loop Fission:

- Attempts to break a single loop into several loops in order to optimize data transfer (behavior main memory, cache and registers)
- Primary objective of optimization is data transfer.

Rule:

- Two statements will be placed into the same loop if there is atleast one variable or array which is referred by both.

Loop Fusion:

- It transforms two adjacent loops into one on the basis of information obtained from data-dependencies analysis.

Remark : Loop Fission and Loop Fusion are related techniques to Strip mining and loop collapsing

Source : Reference [4],[6], [7], [8], [32], 34]

Loop Fusion

(Contd…)

- It is merging of several loops into a single loop
  - Example : **Untuned**

```
for(i=0; i < 100000; i++)
    x = x * a[i] + b[i];
for(i=0; i < 100000; i++)
    y = y * a[i] + c[i];
```

Example : Tuned

```
for(i=0; i < 100000; i++) {
    x = x * a[i] + b[i];
    y = y * a[i] + c[i];
 }
```

- Tuned code runs atleast 10 times faster on Ultra Sparc (both with –O3 flag)

Loop Fusion

(Contd…)

- **Advantages**
  - The loop overhead is reduced by a factor of two in the above case.
  - Allows for better instruction overlap in loops with dependencies.
  - Cache misses can be decreased if both loops reference the same array.
- **Disadvantages**
  - Has the potential to increase cache misses if the fused loops contain references to more than four arrays and the starting elements of those arrays map to the same cache line.

    e.g:

    ```
    x = x * a[i] + b[i] * c[i] + d[i] / e[i]
    ```

Loop Optimizations : Basic Loop Unrolling

- Loop optimizations accomplish three things :
  - Reduce loop overhead
  - Increase Parallelism
  - Improve memory performance patterns
- Understanding your tools and how they work is critical for using them with peak effectiveness. For performance, a compiler is your best friend.
- Loop unrolling is performing multiple loop iterations per pass.
- Loop unrolling is one of the most important optimizations that can be done on a pipelined machine.
- Loop unrolling helps performance because it fattens up a loop with calculations that can be done in parallel
- **Remark :** Never unroll an inner loop.

Qualifying Candidates for Loop Unrolling

- The previous example is an ideal candidate for loop unrolling.
- Study categories of loops that are generally not prime candidates for unrolling.
  - Loops with low trip counts
  - Fat loops
  - Loops containing branches
  - Recursive loops
  - Vector reductions
- To be effective, loop unrolling requires that there be a fairly large number of iterations in the original loop.
- When a trip count in loop is low, the preconditioning loop is doing proportionally large amount of work.

Qualifying candidates for Loop Unrolling

(Contd..)

❖ **Loop containing procedure calls**

Loop containing subroutine or function calls generally are not good candidates for unrolling.

- **First :** They often contain a fair number of instructions already. The function call can cancel many more instructions.

- **Second :** When the calling routine and the subroutine are compiled separately, it is impossible for the compiler to intermix instructions.

- **Last :** Function call overhead is expensive. Registers have to be saved, argument lists have to be prepared.The time spent calling and returning from a subroutine can be much greater than that of the loop overhead.

Source : Reference [4],[6], [7], [8], [32], 34]

Qualifying Candidates for Loop Unrolling

(Contd…)

```
DO 10 I=1, N
   CALL SHORT(A(I), B(I),C)
10 CONTINUE
   SUBROUTINE SHORT (A,B,C)
   A = A+B+C
   RETURN
    END
```

```
      II=IMOD (N,4)
     DO 9 I=1, II
       CALL SHORT (A(I),B(I),C)
9   CONTINUE
     DO 10 I=1+II, N,4
       CALL SHORT(A(I),B(I),C)
       CALL SHORT(A(I+1),B(I+1),C)
       CALL SHORT(A(I+2),B(I+2),C)
       CALL SHORT(A(I+3),B(I+3),C)
10 CONTINUE
```

- If a particular loop is already fat, then unrolling is not going to help much and loop overhead will spread over a fair number of instructions.
  - A good rule of thumb is to look elsewhere for performance when the loop inwards exceed three or four statements.
  - Since code indicates that in lining is feasible.

Qualifying Candidates for Loop Unrolling

(Contd..)

❖Recursive Loops

Original

```
DO 10 I=2, N
   A(I) = A(I) + A(I-1) x B
10 CONTINUE
```

Modified

```
A(I)   = A(I)+A(I-1)*B
A(I+1) = A(I+1)+A(I)*B
A(I+2) = A(I+2)+A(I+1)*B
A(I+3) = A(I+3)+A(I+2)*B
```

- Dependency can be reduced by deriving new set of recursive equations
- Decreasing the dependencies at the expense of creating more work.

```
DO 10 I =2, N,2
   A(I) = A(I+1) + A(I-1) * B +  A(I-1) *B*B
   A(I) = A(I) + A(I-1)*B
10 CONTINUE
```

- This is an example of vector recursion-though very to improve.

A Good compiler can make the rolled up version go faster by recognizing the dependency as opportunity to save memory traffic.

Outer and Inner Loop Unrolling

Remark : The loop or loops in the center are called the *inner loops and* the surrounding loops are called outer loops

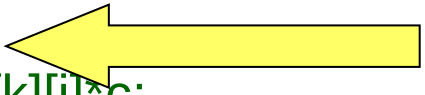
- Loopnest: Enabled loops within other created loops

- Original: for loop nest

```
for (i=0;  i<n;  i++)
for (j=0; j<n;  j++)
                for (k=0; k<n; k++)
      a[i][j][k] = a[i][j][k] + b[i][j][k]*c;
```

- Unrolling the middle (j) loop twice

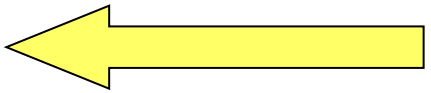
```
for (i=0; i<n; i++)
for (j=0; j<n; j+=2)
for (k=0; k<n; k++){
        a[i][j][k] = a[i][j][k]   + b[i][k][j]*c;
a[i][j+1][k] = a[i][j+1][k] + b[i][k][j+1]*c;

}
```



Modified code

Outer and Inner Loop Unrolling

- Unrolling the ‘k’ loop twice



Modified code

```
for(I =0; i<n; i++)
for(j = 0; j<n; j+=2)
     for(k = 0; k<n;  k+=2){
       a[i][j][k]     = a[i][j][k]    + b[i][k][j]*c;
   a[i][j+1][k]     = a[i][j+1][k]       + b[i][k][j+1]*c;
   a[i][j][k+1]     = a[i][j][k+1]       + b[i][k+1][j]*c;
   a[i][j+1][k+1]  = a[i][j+1][k+1]   + b[i][k+1][j+1]*c;

}
```

- Reasons for applying outer loop unrolling are:
  - To expose more computations
  - To improve memory reference patterns

Loop Unrolling and Sum Reduction

- Loop Unrolling should be used to reduce data dependency. Different variables can be used to eliminate the data dependency
  - Untuned Loop

```
a=0.0;
for (I=0;   i<ARRAY_SIZE; i++)
   for (j=0; j< ARRAY_SIZE;j++)
          a = a+ b[j]*c[i];
```

Original code

  - Tuned Loop (unrolled to depth 4)

```
a1 = a2 = a3 = a4 = 0.0;
for (i=0; i<ARRAY_SIZE; i++)
for (j=0; j< ARRAY_SIZE; j++){
a1 = a1 + b[j]  *c[i];
a2 = a2 + b[j+1]*c[i];
a3 = a3 + b[j+2]*c[i];
a4 = a4 + b[j+3]*c[i];
aa = a1 + a2 + a3 + a4; }
```

Modified code

- Speed increased by a factor of 4 ! (with appropriate compiler switches))

Loop Interchange

- Loop interchange is a technique for rearranging a loop nest so that the right stuff at the center. What is the right stuff depends upon what you are trying to accomplish.
- Loop interchange to move computations to the center of the loop nest.
- It is also good for improving memory access patterns.
- Iterations on the wrong subscript can cause a large stride and hurt your performance.
- Inverting the loops, so that the iterating variables causing the lesser strides are in the center, you can get performance win.

Source : Reference [4],[6], [7], [8], [32], 34]

Loop Interchange

(Contd…)

- Loop interchange to move computations to the center
- Frequently, the interchange of nested loops permits a significant increase in the amount of parallelism
- Example is straight forward: it is easy to see that there are no inter iteration dependencies.

```
PARAMETER(IDIM=1000,JDIM=1
000                              ,
KDIM=4)
DO 10  I =1, IDIM
      DO 20 J  =1, JDIM
         DO 30 K =1, KDIM
         D(K,J,I)=D(K,J,I)+
V(K,J,I)*DT
30 CONTINUE
20 CONTINUE
10 CONTINUE
```

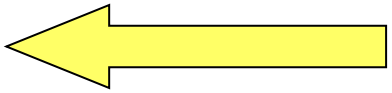
```
PARAMETER(IDIM=1000,JDIM=1000,
                  KDIM = 4)

 DO 10 K=1, IDIM
   DO 20 I=1, KDIM
    DO 30 J=1, JDIM
     D(K,J,I)=D(K,J,I)+ V(K,J,I)*DT
30 CONTINUE
20 CONTINUE
10 CONTINUE
```

Loop Interchange

(Contd…)

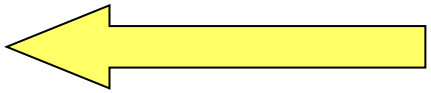
- Loop Interchange is done to minimize the stride access corresponding to array elements in the innermost loops.
- Interchanging loops can also reduce the loop overhead when the inner loop are iterate much less than the outer loops

```
float a[2000][40][2]
for(i=0; i<2000; i++)
for(j=0; j<40; j++)
for(k=0; k<2; k++)
        a[i][j][k] = a[i][j][k] * 2.50 + 0.056;
```



Original code

```
float a[2][40][2000]
for(i=0; i<2; i++)
for(j=0; j<40; j++)
for(k=0; k<2000; k++)
        a[i][j][k] = a[i][j][k] * 2.50 + 0.056;
```



Modified code

- A reduction of about 15 % execution time was obtained in C/Fortran

Loop Optimization: Invariant Code Extraction

- Statements that do not change within an inner loop can be moved outside of the loop. (Compiler optimizations can usually detect these).

Example : Data Wrap around, Untuned

```
for(i=0 ; i<1000; i++){
for(j=0 ; j<1000; j++){

 if(a[i]>100) b[i] =  a[i]–5.0;
         x=x+a[j] + b[i];
 }
/*   BEWARE if a[i]=a[i]–5.0;
      result could be very different!*/
```

Data Wrap around, tuned

```
for(i=0;i<1000; i++)
{
     if(a[i]>100)
     b[i]=a[i]–5.0;
for(j=0;j<1000; j++)
 x = x + a[j] + b[i];
}
```

Remark : Tuned code can about 75 times faster than untuned code!

Loop Optimization: Loop De-factorization

- Loop De-factorization consists of removing common multiplicative factors outside of inner loops

Example Factorized

```
for(i=0; i<1000; i++){
   a[i] = 0.0;
for(j=0; j<1000; j++)
a[i] = a[i] + b[j]*d[j]*c[j];}
```

De-Factorized

```
for(i=0; i<1000; i++){
 a[i] = 0.0;
 for(j=0; j<1000; j++){
  a[i] = a[i]+  b[j]*d[j];
  a[i] = a[i]*c[j];}
```

Remark :

- On some platforms, there is benefit in doing this since one (two) multiplication(s) AND one (two) addition(s) can be done simultaneously in one clock cycle!
- Compiler optimizations will not be able to determine that neighbor data dependency.
- Results may vary due to precision of computer

Loop Optimization: IF, WHILE, and DO Loops

- Avoid IF/GOTO loops and WHILE loops. They inhibit compiler optimizations and they introduce unnecessary overheads.

Untuned Loops (IFs and GOTOs):

```
      I=0
10  I = I + I
    IF(I.GT.100000)GOTO 30
    A(I)=A(I) + B(I)*C(I)
     GOTO 10
30 CONTINUE
```

Turned Loop :

```
      I = 0
10 I = I + 1
    A(I)=A(I)+B(I)*C(I)
IF(I.LE.100000)GOTO 10
```

Another Untuned Loop (WHILE Loop) :

```
    I = 0
    DO WHILE (I .LT. 100000)
          I = I + 1
       A(I) = A(I)+B(I)*C(I)
  ENDDO
```

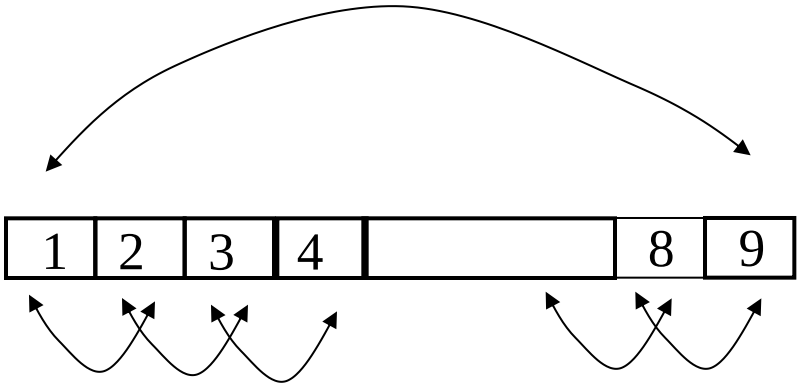
Turned Loop:

```
      DO I = 1, 100000
       A(I) = A(I)+B(I)*C(I)
      END DO
```

Loop Optimization: Neighbor Data Dependency

Example: data wrap around, untuned version

```
jwrap = ARRAY_SIZE –1;
for(i=0; i<ARRAY_SIZE; i++)
  b[i] =(a[i]+a[jwrap])*0.5;
 jwrap = i; }
```



- Compiler optimizations will not be able to determine that a [`jwrap`] is a neighbor value

Example: data wrap around, tuned version:

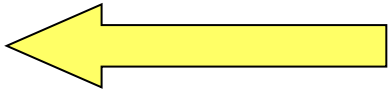
```
b[0] = (a[0] + a[ARRAY_SIZE –1]) * 0.5;
for(i=1 ; i < ARRAY_SIZE ; i++)
 b[i] = (a[i]+a[i-1]) * 0.5;
```

Remark : Once the program is debugged, declare arrays to exact sizes whenever possible. This reduces memory use and also optimizes pipelining and cache utilization.

Programming Techniques – Managing the Cache

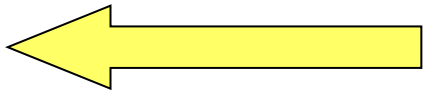
We can modify the previous code to better use the cache.

```
DO 10, J = 1, N
  DO 10, I = 1, N
    DO 10, K = 1, N
      C(I, J) = C (I, J) + A (I, K) * B (K, J)
10 CONTINUE
```



Original code

```
DO 10, JB = 1, N, NB
  DO 10, IB = 1, N, NB
    DO 10, KB = 1, N, NB
      DO 10, J = JB, JB + NB – 1
        DO 10, I = IB, IB + NB – 1
          DO 10, K = KB, KB + NB – 1
            C (I, J) = C (I, J) + A (I, K) * B(K,J)
10 CONTINUE
```



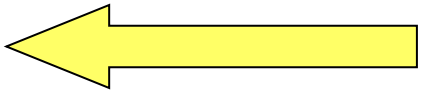
Modified code

This is most useful as a simple example of cache blocking. Compilers cache block the original code as part of ordinary optimization.

Programming Techniques –Cache Blocking

- Cache blocking is most effective at the highest level in the code.
- Even code that uses cache-blocked routines in tuned system math libraries can sometimes be blocked at a higher level.
- Consider the following code to compute C = AB and E = AD for N x N matrices A, B, C, D, and E:

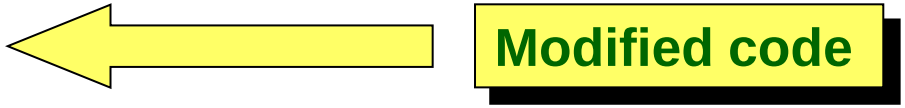
```
      CALL DGEMM('NO TRANSPOSE A', 'NO TRANSPOSE B', N, N, N,
     $  1.0D0, A, LDA, B, LDB, 0.ODO, C, LDC)

      CALL DGEMM('NO TRANSPOSE B', 'NO TRANSPOSE A', N, N, N, N,
     $   1.0D0, A, LDA, D, LDD, 0.D0, E, LDE)
```



Original code

Programming Techniques –Cache Blocking

This is the globally blocked code from the previous example:




```
DO 10, J = 1, N, NB

   DO 10, K = 1, N, NB

      CALL DGEMM ('NO TRANS A', 'NO TRANS B', N, NB, NB,
$       1.0D0, A(1, K), LDA, B(K,J), LDB, 0.0D0,   C(1,J), LDC)


      CALL DGEMM ('NO TRANS A', 'NO TRANS D', N, NB, NB,
$      1.0D0, A(1, K), LDA, D(K,J), LDD, 0.ODO, E(1,J), LDE)

10 CONTINUE
```

Source : Reference [4],[6], [7], [8], [32], 34]

Conclusions

- Reducing Memory Overheads is important for performance on Multi cores
- Minimization of memory traffic is the single most important goal.
- Advanced Compiler Optimization flags can be used for performance
- Write code so that a compiler find it easy to locate optimizations
- Reduce the Overheads due to Multi-Threaded Programming.

References

1. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Programming, Boston, MA : Addison-Wesley

2. Butenhof, David R **(1997),** Programming with POSIX Threads , Boston, MA : Addison Wesley Professional

3. Culler, David E., Jaswinder Pal Singh **(1999),** Parallel Computer Architecture - A Hardware/Software Approach , San Francsico, CA : Morgan Kaufmann

4. Grama Ananth, Anshul Gupts, George Karypis and Vipin Kumar **(2003),** Introduction to Parallel computing, Boston, MA : Addison-Wesley

5. Intel Corporation, **(2003),** Intel Hyper-Threading Technology, Technical User's Guide, Santa Clara CA : Intel Corporation Available at : http://www.intel.com

6. Shameem Akhter, Jason Roberts **(April 2006),** Multi-Core Programming - Increasing Performance through Software Multi-threading , Intel PRESS, Intel Corporation,

7. Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell **(1996),** Pthread Programming O'Reilly and Associates, Newton, MA 02164,

8. James Reinders, Intel Threading Building Blocks – (**2007**) , O'REILLY series

9. Laurence T Yang & Minyi Guo (Editors), (**2006**) *High Performance Computing - Paradigm and Infrastructure* Wiley Series on Parallel and Distributed computing, Albert Y. Zomaya, Series Editor

10. Intel Threading Methodology ; Principles and Practices Version 2.0 copy right **(March 2003),** Intel Corporation

References

11. William Gropp, Ewing Lusk, Rajeev Thakur **(1999),** Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press..

12. Pacheco S. Peter, **(1992),** Parallel Programming with MPI, , University of Sanfrancisco, Morgan Kaufman Publishers, Inc., Sanfrancisco, California

13. Kai Hwang, Zhiwei Xu, (**1998**), Scalable Parallel Computing (Technology Architecture Programming), McGraw Hill New York.

14. Michael J. Quinn (**2004**), Parallel Programming in C with MPI and OpenMP McGraw-Hill International Editions, Computer Science Series, McGraw-Hill, Inc. Newyork

15. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Progrmaming, Boston, MA : Addison-Wesley

16. SunSoft. Solaris multithreaded programming guide. SunSoft Press, Mountainview, CA, **(1996),** Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill,

17. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, **(2001)**,Parallel Programming in OpenMP San Fracncisco Moraan Kaufmann

18. S.Kieriman, D.Shah, and B.Smaalders **(1995),** Programming with Threads, SunSoft Press, Mountainview, CA. 1995

19. Mattson Tim, **(2002),** Nuts and Bolts of multi-threaded Programming Santa Clara, CA : Intel Corporation, Available at : http://www.intel.com

20. I. Foster **(1995,** Designing and Building Parallel Programs ; Concepts and tools for Parallel Software Engineering, Addison-Wesley (1995)

21. J.Dongarra, I.S. Duff, D. Sorensen, and H.V.Vorst **(1999),** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) SIAM, 1999

References

22. OpenMP C and C++ Application Program Interface, Version 1.0". **(1998),** OpenMP Architecture Review Board. October 1998

23. D. A. Lewine. *Posix Programmer's Guide:* **(1991),** Writing Portable Unix Programs with the Posix. 1 Standard. O'Reilly & Associates, 1991

24. Emery D. Berger, Kathryn S McKinley, Robert D Blumofe, Paul R.Wilson, *Hoard : A Scalable Memory Allocator for Multi-threaded Applications* ; The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). Cambridge, MA, November (**2000)**. Web site URL : http://www.hoard.org/

25. Marc Snir, Steve Otto, Steyen Huss-Lederman, David Walker and Jack Dongarra, (**1998**) *MPI-The Complete Reference: Volume 1, The MPI Core, second edition* [MCMPI-07].

26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (**1998**) *MPI-The Complete Reference: Volume 2, The MPI-2 Extensions*

27. A. Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill, **(1996)**

28. OpenMP C and C++ Application Program Interface, Version 2.5 (**May 2005**)", From the OpenMP web site, URL **: http://www.openmp.org/**

29. Stokes, Jon 2002 Introduction to Multithreading, Super-threading and Hyper threading *Ars Technica*, October **(2002)**

30. Andrews Gregory R. 2000, Foundations of Multi-threaded, Parallel and Distributed Programming, Boston MA : Addison – Wesley (**2000)**

31. Deborah T. Marr , Frank Binns, David L. Hill, Glenn Hinton, David A Koufaty, J . Alan Miller, Michael Upton, "Hyperthreading, Technology Architecture and Microarchitecture", Intel (**2000-01)**

References

32. Kevin Dowd, *O'Reilly & Associates, Inc.,First edition, ISBN: 1-56592-032-5 (1993)*

33. Rajat P. Garg and Ilya Sharapov *"Techniques for Optimizing Applications: High Performance Computing"* ISBN: 0-13-093476-3 , 2002

34. Dr.Christian Halloy and Dr. Kwai Wong *"Parallel Computing Techniques to Maximize Your Megaflops"* upercomputing'99-Portland, OR Tutorial Workshop Notes, November 15, 1999

Thank You

Any questions ?