

A Parallel Approach for Matrix Multiplication on the TMS320C4x DSP

Application Report

Rose Marie Piedra
Digital Signal Processing — Semiconductor Group

SPRA107
February 1994



IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Introduction

Matrix operations, like matrix multiplication, are commonly used in almost all areas of scientific research. Matrix multiplication has significant application in the areas of graph theory, numerical algorithms, signal processing, and digital control.

With today's applications requiring ever higher computational throughputs, parallel processing is an effective solution for real-time applications. The TMS320C40 is designed for these kinds of applications.

This application note shows how to achieve higher computational throughput via parallel processing with the TMS320C40. Although the focus is on parallel solutions for matrix multiplication, the concepts stated here are relevant to many other applications employing parallel processing.

The algorithms that are presented were implemented on the Parallel Processing Development System (PPDS), which has four TMS320C40s and both shared- and distributed-memory support. The algorithms make use of parallel-runtime-support library (PRTS) functions available with the 'C40 C compiler for easy message passing.

This report is structured in the following way:

<i>Matrix Multiplication</i>	Gives a brief review of matrix multiplication and some common application areas.
<i>Fundamentals of Parallel Processing</i>	Presents some basic concepts of parallel processing. Partitioning, memory configuration, interconnection topologies, and performance measurements are some of the issues discussed.
<i>Parallel Matrix Multiplication</i>	Focuses on parallel implementations of matrix multiplication. Shared- and distributed-memory implementations are considered, as well as TMS320C40 suitability for each.
<i>Results of Matrix Multiplication on a TMS320C40-Based Parallel System</i>	Presents the results of shared- and distributed-memory implementations of parallel matrix multiplication on the 'C40 PPDS. Includes analysis of speed-up, efficiency, and load balance.
<i>Conclusion</i>	States conclusions.
<i>Appendices</i>	List the code for parallel matrix multiplication. The programs have been written in C. For faster execution, a C-callable assembly language routine is also supplied.

Matrix Multiplication

Let A and B be matrices of size $n \times m$ and $m \times l$, respectively. The product matrix $C = A * B$ is an $n \times l$ matrix, for which elements are defined as follows [7]:

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj} \quad \text{where} \quad 0 \leq i < n, 0 \leq j < l$$

The matrix multiplication requires $O(nml)$ arithmetic operations, with each arithmetic operation requiring a cumulative multiply-add operation. When $l=1$, a matrix-vector multiplication exists. Assuming that $n=m=l$, matrix multiplication is an $O(n^3)$ operation.

Matrix-multiplication applications range from systems-of-equations solutions to graph representation. Also, matrix-vector multiplication can be applied to compute linear convolution. Refer to [2] and [7] for further information on these techniques.

Fundamentals of Parallel Processing

When applications require throughput rates that are not easily obtained with today's sequential machines, parallel processing offers a solution.

Generally stated, parallel processing is based on several processors working together to accomplish a task. The basic idea is to break down, or partition, the computation into smaller units that are distributed among the processors. In this way, computation time is reduced by a maximum factor of p , where p is the number of processors present in the multiprocessor system.

Most parallel algorithms incur two basic cost components[7]:

- computation delay—under which we subsume all related arithmetic/logic operations, and
- communication delay—which includes data movement.

In a realistic analysis, both factors should be considered.

This application report presents some basic concepts of parallel processing. Refer to [2], [4], [5], [6], and [7] for more detailed information.

Partitioning Schemes

From the software point of view, two basic approaches are used to create a parallel application:

- **Functional Partitioning:** In this case, the task is a single function that has been subdivided between the processors. Each processor performs its subfunction on the data as it moves from one processor to the next in an assembly line or pipeline fashion.
- **Data Partitioning:** In this case, the task is partitioned so that each processor performs exactly the same function, but on different subblocks of the data. This approach requires algorithms with strong intrinsic parallelism. The parallel matrix multiplication implemented with the TMS320C40 PPDS applies this data-partitioning approach.

Architectural Aspects

From the hardware point of view, two important issues should be considered:

- **Memory configuration (shared- versus distributed-memory):** In a distributed-memory system, each processor has only local memory, and information is exchanged as messages between processors. In contrast, the processors in a shared-memory system share a common memory. Although data is easily accessible to any processor, memory conflict constitutes the bottleneck of a shared-memory configuration. Because the PPDS has both shared and distributed memory, it is an excellent tool for implementing and evaluating different parallel configurations.
- **Connectivity network:** This issue relates to the way the processors are interconnected with each other. Fully connected networks (in which all the processors are directly connected to each other) are the ideal networks from an “ease of use” point of view. However, they are impractical in large multiprocessor systems because of the associated hardware overhead. Linear arrays, meshes, hypercubes, trees, and fully-connected networks are among the topologies most commonly

used. Hypercube topologies are widely popular in commercially available multiprocessor systems because they provide higher connectivity and excellent mapping capabilities. In fact, it is possible to embed almost any other topology in a hypercube network [4]. Mesh topologies are also commonly used to make systems modular and easily expandable. When distributed-memory systems are used, interconnectivity issues play an important role in the message-passing mechanism.

Performance Measurements

Two measurements apply to the performance of a parallel algorithm—speed-up and efficiency.

- **Speed-up** of a parallel algorithm is defined as $S_p = T_s/T_p$, where T_s is the algorithm execution time when the algorithm is completed sequentially, and T_p is the algorithm execution time using p processors. Theoretically, the maximum speed-up that can be achieved by a parallel computer with p identical processors working concurrently on a single problem is p . However, other important factors (such as the natural concurrence in the problem to be computed, conflicts over memory access, and communication delay) must be considered. These factors can reduce the speed-up.
- **Efficiency**, defined as $E_p = S_p/p$ with values between (0,1), is a measure of processor utilization in terms of cost efficiency. An efficiency close to 1 reveals an efficient algorithm. If the efficiency is lower than 0.5, it is often better to use fewer processors because using more processors offers no advantage.

Generally, the communication cost should be minimized by using wiser partitioning schemes and by overlapping CPU and I/O operations. DMA channels help to alleviate the communication burden.

Parallel Matrix Multiplication

In parallel matrix multiplication, successive vector inner products are computed independently.

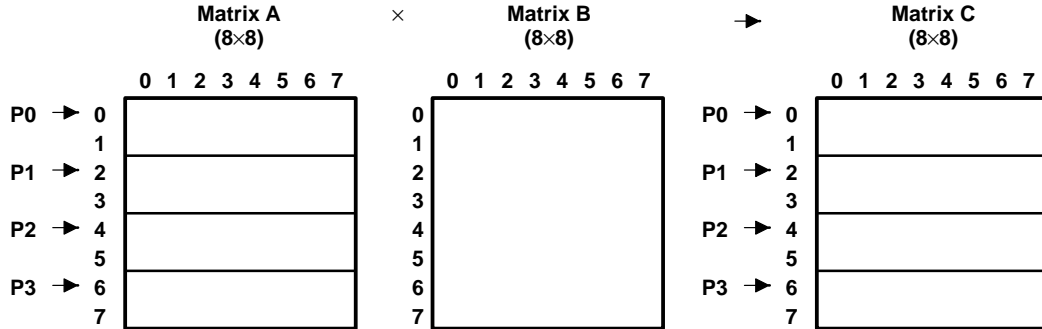
Because this application report focuses on multiple instruction multiple data (MIMD) implementations (shared- and distributed-memory approaches), systolic implementations are not discussed. However, single instruction multiple data (SIMD) implementations are also feasible with the TMS320C40.

Shared-Memory Implementation

Let $n = qp$, where n is the number of rows of matrix A, p is the number of processors, and $q \geq 1$ is an integer.

Matrices A and B are stored in global memory so that each processor can have access to all the rows/columns. The basic idea is to allocate a different working set of rows/columns to each processor. Processor i computes row vectors $qi, qi+1, \dots, qi+q-1$ of product matrix C, where $i = 0, 1, \dots, p-1$. This is illustrated in Figure 1 for $p = 4$ and $n = m = l = 8$.

Figure 1. Shared-Memory Implementation



Note: All processors have full access to the entire matrix B.

Two different approaches can be followed:

1. **Execute operations totally in shared memory (full memory conflict):** This implementation does not require any initial data transfer, but a conflict among memory accesses results (see code in Appendix A).
2. **Transfer data for execution in on-chip memory of each processor (reduced memory conflict):** This approach reduces the delay caused by memory conflicts, but it requires extra data transfer. This moving of data can be executed by the CPU or DMA via double-buffering techniques.

Using double-buffering techniques can minimize the data-transfer delay. For matrix-vector multiplication, vector B is initially transferred to on-chip RAM. While the CPU is working on row $A_{(i)}$, the DMA is bringing row $A_{(i+1)}$ to on-chip RAM. If the two buffers are allocated in different on-chip RAM blocks, no DMA/CPU conflict will be present. If the DMA transfer time is less than or equal to the CPU computation time, the communication delay will be fully absorbed. The TMS320C40 has two 4K-byte on-chip RAM blocks that enable it to support double buffering of up to 1K words. Although this approach is not implemented in this application report, page 6 shows what kind of performance can be expected.

Distributed-Memory Implementation

Let $n = qp$, where n is the number of rows of matrix A, p is the number of processors, and $q \geq 1$ is an integer.

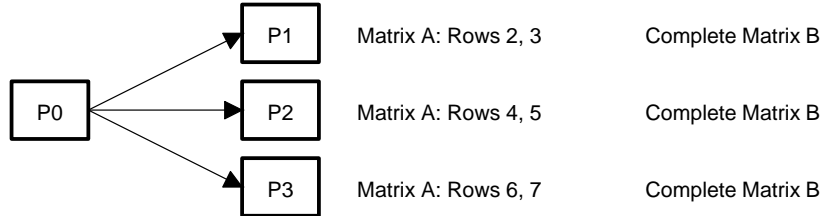
Matrix A has been partitioned into p regions with each region containing q rows and being assigned to the local-memory (LM) of each processor. Matrix B is made available to all the processors. The data-partitioning scheme is similar to the shared-memory approach. The differences are the extra time required for data distribution/collection via message passing and the fact that all computations are done in the LM of each processor with no memory-access conflict involved. With the use of double-buffering, this communication delay can be reduced.

In this implementation, it is assumed that only processor 0 has access to matrix A and B. Processor 0 acts as a host processor responsible for broadcasting the needed data to each of the other processors and waiting for the vector results from the other processors. This is illustrated in Figure 2. Data distribution/collection is system-specific and may not be needed for certain applications.

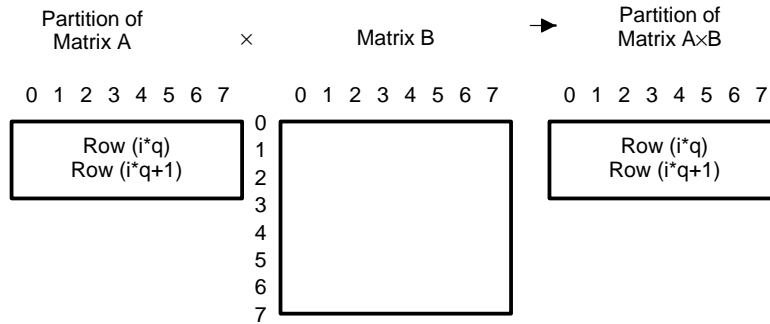
Figure 2. Distributed-Memory Implementation

Step 1: Data Broadcasting (Asynchronous)

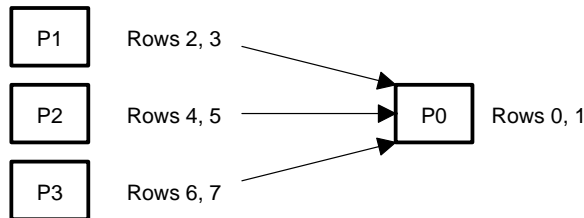
$$q = (n/p) = (8/4) = 2$$



Step 2: Distributed Matrix Multiplication (Processor i)



Step 3: Data Collection (Asynchronous)



Note: Asynchronous = using DMA channels.

TMS320C40 Implementation

The TMS320C40 is the first parallel-processing DSP. In addition to a powerful CPU that can execute up to 11 operations per cycle with a 40- or 50-ns cycle time, it contains 6 communication ports and a multichannel DMA [3]. The on-chip communication ports allow direct (glueless) processor-to-processor communication, and the DMA unit provides concurrent I/O by running parallel to the CPU. Also, special interlocked instructions provide support for shared-memory arbitration. These features make the TMS320C40 suitable for both distributed- and shared-memory computing systems.

Results of Matrix Multiplication on a TMS320C40-Based Parallel System

Parallel matrix multiplication was implemented in the TMS320C40 PPDS. The PPDS is a stand-alone development board with four fully interconnected TMS320C40s. Each 'C40 has 256K bytes of local memory (LM) and shares a 512K-byte global memory (GM)[1].

Features of implementing parallel matrix multiplication in the TMS320C40 PPDS:

- The programs are generic. You can run the programs for different numbers of processors in the system just by changing the value of P (if you set P=1, you will have a serial program).
- Data input is provided in a separate file to preserve the generality of the programs.
- A node ID must be allocated to each processor. In this way, each processor will select automatically the row/column working set allocated to it. In this implementation, a different node ID is allocated to each processor by using the 'C40 debugger commands to initialize that variable. It is also possible to allocate a node ID by using the *my_id* function in the parallel-runtime-support library (PRTS), which reads a predetermined set node ID value from a user-specified memory location.
- For benchmarking of shared-memory programs, a global start of all the processors is absolutely necessary; otherwise, the real-memory-access conflict will not be observed. To help with this process, a C-callable assembly routine is provided in Appendix C (*syncount.asm*) for debugging systems without global start capability. Rotating priority for shared memory access should be selected by setting the PPDS LCSR register to 0x40. On this basis, the total execution time of the parallel algorithm can be defined as $T = \max(T_i)$, where T_i is the execution time taken by processor i (see Appendix A, *shared.c*: T_i = time between labels t_2 and t_1).
- For benchmarking of distributed-memory programs, I/O-execution time is optional. Data I/O is system-specific and normally is not considered. In this application report, speed-up/efficiency figures are given for both cases—including and not including I/O—in order to show the effect of the communication delay in a real application. In this program (see Appendix B, *distrib.c*), when processor 0 is acting as a host, then

Execution time = time between labels t_1 and t_4 in processor 0.
(I/O included)

Execution time = time between labels t_2 and t_3 in the processor with more load, or in any
(I/O not included) processor in the case of load balancing.

- If a debugger with benchmarking options (*runb*) is not available, the 'C40 analysis module or the 'C40 timer can be used. In this application report, the 'C40 timer and the timer routines provided in the PRTS library have been used for speed-up efficiency measures. Serial program timings for the speed-up figures were taken with the shared-memory program with P = 1.
- When the number of rows of matrix A is not a multiple of the number of processors in the system, load imbalance occurs. This case has been considered for the shared-memory (full-memory-conflict) implementation. (See Appendix A, *shared.c*.)

In parallel processing, speed-up/efficiency figures are more important than cycle counting because speed-up/efficiency figures show how much performance improves if you make an application parallel. You can apply the speed-up factors to any known sequential benchmarks to get a rough idea of the parallel-execution time (assuming that the same memory allocation is used). Appendix D includes a

C-callable assembly-language function that executes matrix multiplication in approximately $nrowsa*(5+ncolsb*(6+ncolsa))$ cycles in single-processor execution. This assumes use of program and data in on-chip RAM. It also shows how you can use that function for parallel-processing execution.

Analysis of the Results

The performance of a parallel algorithm depends on the problem size (matrix size in our case) and on the number of processors in the system. Speed-up and efficiency figures covering those issues can be observed from Figure 3 to Figure 8 for the parallel algorithms presented. As you can see:

- Shared-memory (full-memory conflict) has the lowest speed-up and efficiency. However, the initial transfer of data to on-chip memory increases the speed-up, and if double-buffering techniques are used, shared-memory implementation becomes as ideal as the distributed-memory approach.
- Speed-up is proportional to the number of processors. In the shared-memory implementation (reduced-memory conflict) or in the distributed case (computation only), an optimal speed-up of p can be reached. See Figure 3 and Figure 5. This result occurs because matrix multiplication does not require any intermediate communication steps. In Figure 4, when $p = 3$, there is a decline in efficiency due to load imbalance.
- In general, efficiency is a better measure to analyze a parallel algorithm because it is more meaningful in processor utilization. For example, compare Figure 3 and Figure 4—the efficiency figure shows more clearly how increasing the number of processors negatively affects the performance of the shared-memory (full-memory conflict) implementation.
- Speed-up/efficiency increases for larger matrices in all cases, except for the shared-memory (full-memory conflict) case. In the distributed-memory case (with I/O), speed-up/efficiency increases because the communication delay ($O(n^2)$ operation) becomes negligible against the computation delay ($O(n^3)$ operation) for large n . See Figure 6 and Figure 7.
- In the case of load imbalance, efficiency decreases because computation is not evenly distributed among the processors. This is plotted in Figure 8. As you can see, if $P = 4$, the worst case occurs when matrix size = 5, because while processor 0 is calculating the last row (row 4), all the other processors are idle. The results shown here were taken for the shared-memory implementation but are applicable for the distributed case.
- The shared-memory implementation requires $n*m+m*l+n*l$ words of shared memory (for matrices A, B, and C, respectively). When this amount of memory is not available in the system, intermediate-file downloading can be used. Another option is in-place computation ($A * B \rightarrow A$) using one intermediate buffer of size n per processor. For the distributed-memory case, the performance depends on the way you implement your initial data distribution. In the application, processor 0 requires $n*m+m*l+n*l$ words of local memory. The other processors require $q*m+m*l+q*l$ of local memory, where $q = \lfloor n/p \rfloor$.

The programs in Appendices A and B have been used to calculate the speed-up/efficiency figures. For the assembly-language case (Appendix D), the speed-up figures for the computation timing are still valid. For the total timing (I/O included) using the assembly-language routine, the C implementation of the PRTS routines lowers the speed-up, but for larger matrices, this is minimized.

Figure 3. Speed-Up Vs. Number of Processors (Shared Memory)

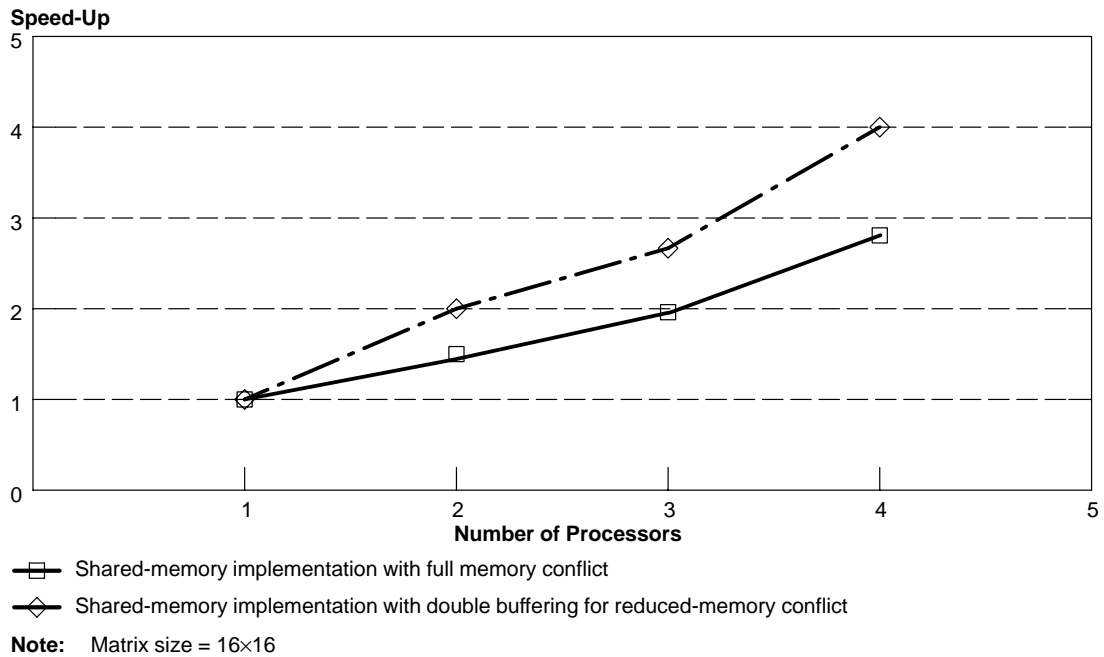


Figure 4. Efficiency Vs. Number of Processors (Shared Memory)

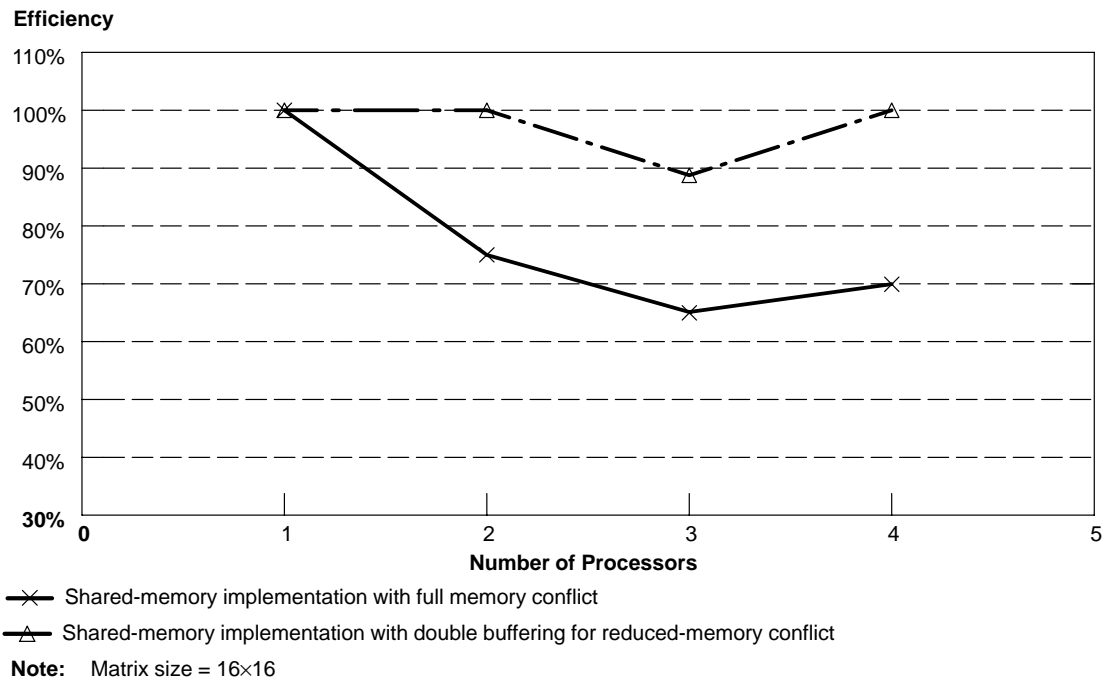


Figure 5. Speed-Up Vs. Number of Processors (Distributed Memory)

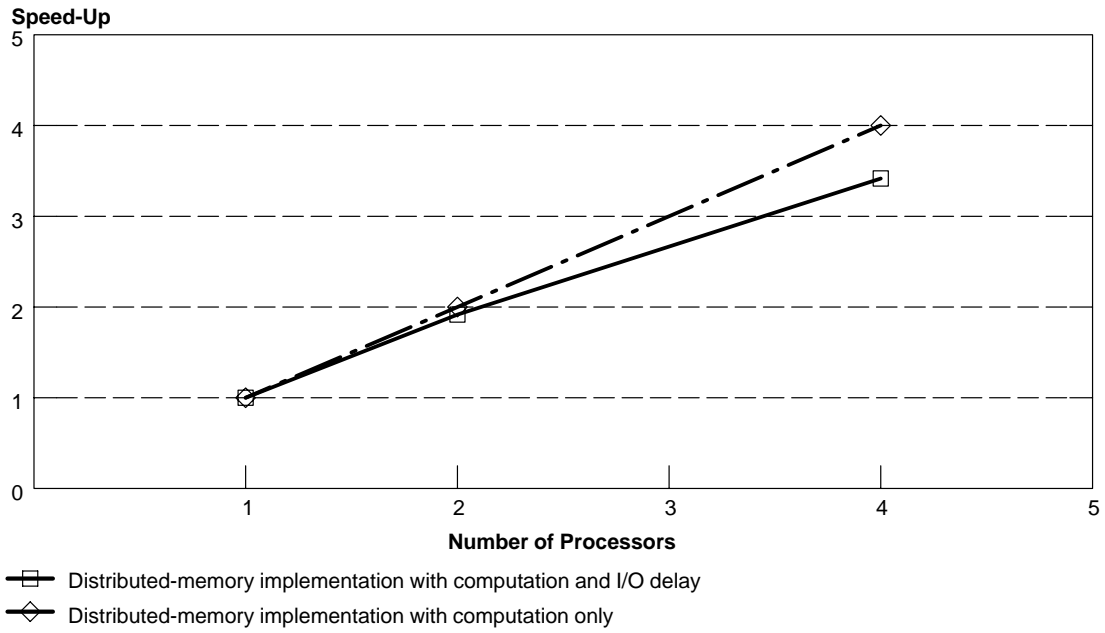


Figure 6. Speed-Up Vs. Matrix Size

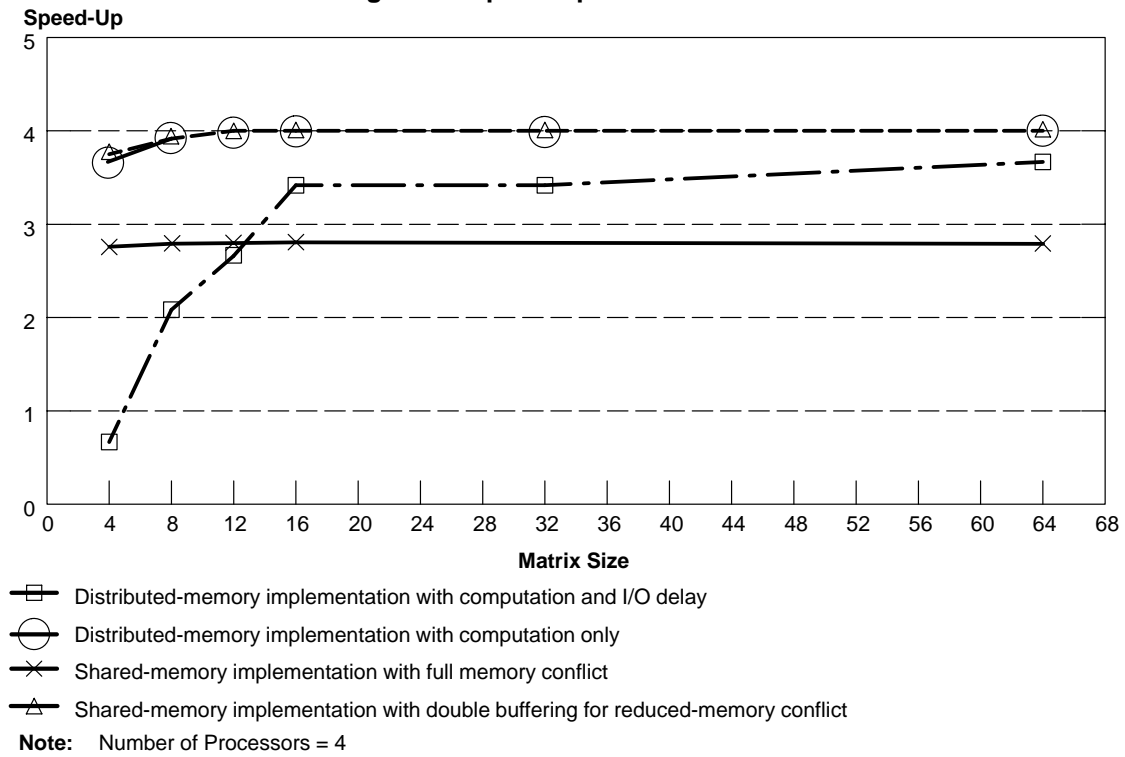
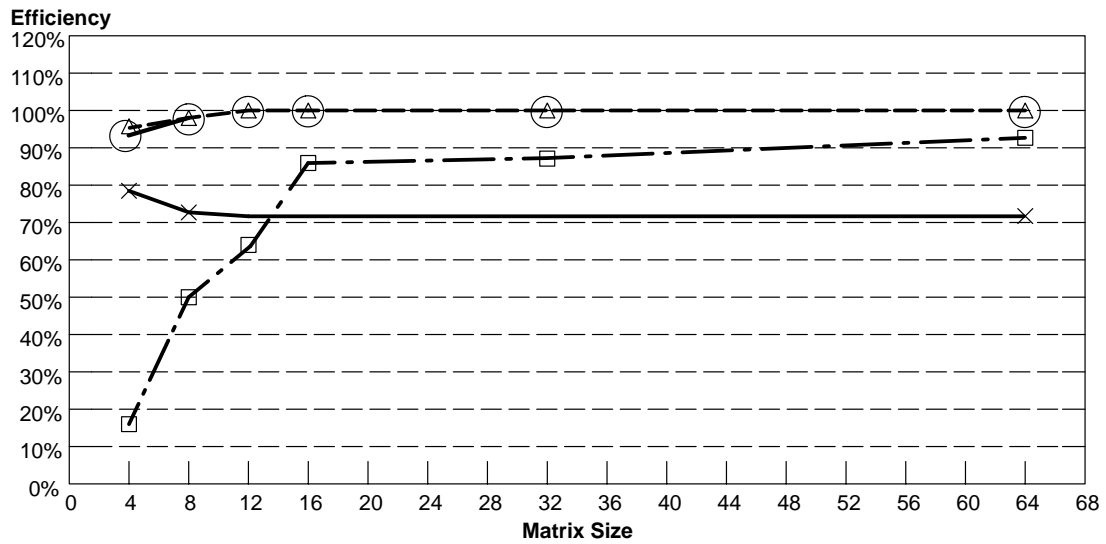


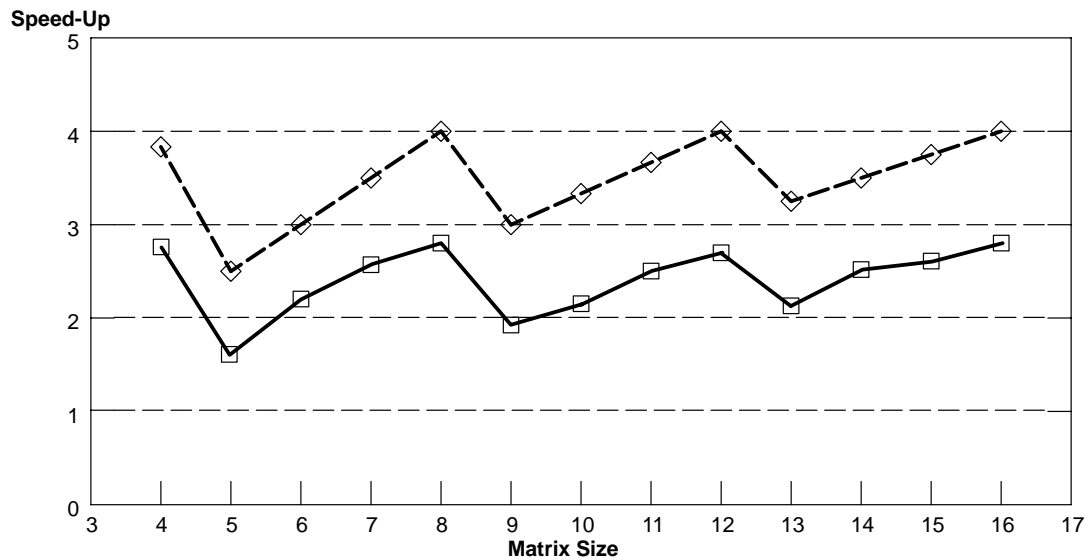
Figure 7. Efficiency Vs. Matrix Size



- Distributed-memory implementation with computation and I/O delay
- Distributed-memory implementation with computation only
- × Shared-memory implementation with full memory conflict
- △ Shared-memory implementation with double buffering for reduced-memory conflict

Note: Number of Processors = 4

Figure 8. Speed-Up Vs. Matrix Size (Load Imbalance for Shared-Memory Program)



- Shared-memory implementation with full memory conflict
- ◇ Shared-memory implementation with double buffering for reduced-memory conflict

Note: Number of Processors = 4

Conclusion

This report has presented parallel implementations of matrix multiplication using both shared- and distributed-memory approaches. Matrix multiplication is an excellent algorithm for parallel processing, as the speed-up/efficiency figures have shown. To avoid memory conflict when using the shared-memory approach, it is important to transfer the data to on-chip/local memory for execution. Because interprocessor communication is required only initially, it does not have a strong effect on the performance of the distributed-memory approach; but with double-buffering techniques, this can be minimized even more. Load balancing must also be considered.

References

- [1] D.C. Chen and R. H. Price. "A Real-Time TMS320C40-Based Parallel System for High Rate Digital Signal Processing." *ICASSP91 Proceedings*, May 1991.
- [2] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs, New Jersey: Prentice-Hall, 1989, page 171.
- [3] *TMS320C4x User's Guide*, Texas Instruments, Incorporated, 1991.
- [4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods*, Englewood Cliffs, New Jersey: Prentice-Hall, 1989.
- [5] S. Y. Kung. *VLSI Array Processors*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.
- [6] U. Schendel. *Introduction to Numerical Methods for Parallel Computers*. England: John Wiley & Sons, 1984.
- [7] J. J. Modi. *Parallel Algorithms and Matrix Computation*. New York: Oxford University Press, 1988.

Appendix A: Shared-Memory Implementation

INPUT0.ASM

```
*****
*
*   INPUT0.ASM: Contains matrix A and B input values.
*
*****

    .global _MAT_A
    .global _MAT_B
    .global _MAT_AxB
    .global _synch      ; counter for synchronization (global start)

    .data
_synch .int    0

_MAT_A      ; stored by rows
    .float   1.0, 2.0, 3.0, 4.0
    .float   5.0, 6.0, 7.0, 8.0
    .float   9.0, 10.0, 11.0, 12.0
    .float   13.0, 14.0, 15.0, 16.0

_MAT_B      ; stored by rows
    .float   1.0, 2.0, 3.0, 4.0
    .float   1.0, 2.0, 3.0, 4.0
    .float   1.0, 2.0, 3.0, 4.0
    .float   1.0, 2.0, 3.0, 4.0

_MAT_AxB .space 16      ; must produce (by rows):
                        ; 10,20,30,40
                        ; 26,52,78,104
                        ; 42,84,126,168
                        ; 58,116,174,232

    .end

/*****
```

SHARED.C

```
/*
SHARED.C : Parallel matrix multiplication (Shared memory version: full memory
conflict)
```

- All the matrices (A,B,C) are stored by rows.

To run:

```
cl30 -v40 -g -o2 -as -mr shared.c
asm30 -v40 -s input0.asm
asm30 -v40 -s syncount.asm
lnk30 shared.obj input0.obj shared.cmd
*****/
#define NROWSA 4 /* number of rows in mat A */
#define NCOLSA 4 /* number of columns in mat A */
#define NCOLSB 4 /* number of columns in mat B */
#define P 4 /* number of processors */

extern float MAT_A[NROWSA][NCOLSA];
extern float MAT_B[NCOLSA][NCOLSB];
extern float MAT_AxB[NROWSA][NCOLSB];

extern int synch; /* synchronization for global start */
extern void syncount();

float *A[NROWSA], *B[NCOLSA], *AxB[NROWSA], temp;

int *synch_p = &synch,
q = NROWSA/P,
ll = 0,
my_node, i, j, k, tcomp;
/*****/

main()
{
asm(" OR 1800h,st"); /* cache enable */

/* accesing matrices declared in an external assembly file */
for (i=0;i<NROWSA;i++) A[i] = MAT_A[i];
for (i=0;i<NCOLSA;i++) B[i] = MAT_B[i];
for (i=0;i<NROWSA;i++) AxB[i] = MAT_AxB[i];

syncount(synch_p,P); /* global start:loop until counter=P */

if (((i = NROWSA %P) >0)) { /* load imbalancing:optional */
if (my_node<i) ++q; else ll = i;
}
ll += q*my_node; /* select beginning of row working set
for processor "my_node" */

t1: time_start(0); /* benchmarking with C40 timer */

for (i=ll;i<(ll+q);i++) /* matrix multiplication */
for (j=0;j<NCOLSB;j++)
{
temp = 0;
for (k=0;k<NCOLSB;k++) temp += A[i][k] * B[k][j] ;
AxB[i][j] = temp;
}

t2 : tcomp = time_read(0); /* shared-memory benchmark */
syncount(synch_p,2*P); /* optional: if you want all processors
finish at the same time */
} /*main*/
```


SHARED.CMD

```

/*****
SHARED.CMD: Linker Command File for Shared-Memory Program
*****/

syncount.obj
-c                               /* link using C conventions */
-stack 0x0100
-lrts40r.lib                     /* get run-time support */
-lprts40r.lib
-m a.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    ROM:    org = 0x00          len = 0x1000
    RAM0:   org = 0x0002ff800   len = 0x0400   /* RAM block0 */
    RAM1:   org = 0x0002ffc00   len = 0x0400   /* RAM block1 */
    LM:     org = 0x040000000    len = 0x10000  /* local memory */
    GM:     org = 0x080000000    len = 0x20000  /* global memory */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .text:    {} > RAM0        /* code */
    .cinit:   {} > RAM1        /* initialization tables */
    .stack:   {} > RAM0        /* system stack */
    .bss :    {} > RAM1        /* global & static vars */
    .data:    {} > GM          /* for input matrix */
}

```

Appendix B: Distributed-Memory Implementation

INPUT.ASM

```
*****
*
*   INPUT.ASM :   Input file for processors 1 to (P-1)
*
*****

        .global    _MAT_A
        .global    _MAT_B
        .global    _MAT_AxB

        .data

_MAT_A   .space 16
_MAT_B   .space 16
_MAT_AxB .space 16

        .end
```

DISTRIB.C

```
/******
DISTRIB.C : Parallel matrix multiplication (distributed-memory
implementation)
            (no load imbalancing has been considered)

cl30 -v40 -g -mr -as -o2 distrib.c
asm30 -v40 -s input0.asm      (see Input0.asm on page 13 )
asm30 -v40 -s input.asm
lnk30 distrib.obj input0.obj distrib.cmd -o a0.out (For processor 0)
lnk30 distrib.obj input.obj distrib.cmd -o a.out (For processors 1 to (P-1))

*****/
#define NROWSA      4          /* number of rows in mat A */
#define NCOLSA      4          /* number of columns in mat A */
#define NCOLSB      4          /* number of columns in mat B */
#define P           4          /* number of processors */

extern float MAT_A[NROWSA][NCOLSA];
extern float MAT_B[NCOLSA][NCOLSB];
extern float MAT_AxB[NROWSA][NCOLSB];

float *A[NROWSA], *B[NCOLSA], *AxB[NROWSA], temp;

int my_node ,
    q = NROWSA/P,
    tcomp, ttotal,
    i,j,k,ll;

int port[4][4] = { 0,0,4,3,
                   3,0,0,4,
                   1,3,0,0,
                   0,1,3,0 }; /* connectivity matrix: processor i is
                                connected to processor j thru port[i][j]:
                                system specific PPDS */

/******
```

```

main()
{
asm(" OR 1800h,st");
/* accessing assembly variables */
for (i=0;i<NROWSA;i++) A[i] = MAT_A[i];
for (i=0;i<NCOLSA;i++) B[i] = MAT_B[i];
for (i=0;i<NROWSA;i++) AxB[i] = MAT_AxB[i];

t1: time_start(0);
/* Processor 0 distributes data. Other processors receive it */

if (my_node==0)
    for(i=1;i<P;++i){ /* asynchronous sending (DMA) */
        send_msg(port[0][i],&A[i*q][0],(q*NCOLSA),1);
        send_msg(port[0][i],&B[0][0],(NCOLSA*NCOLSB),1);
/* autoinitialization can also be used */
    }
else { /* synchronous receiving (CPU) */
    k = in_msg(port[my_node][0],&A[0][0],1);
    k = in_msg(port[my_node][0],&B[0][0],1);
}

t2: tcomp = time_read(0);

for (i=0;i<q;i++) /* Matrix multiplication */
for (j=0;j<NCOLSB;j++)
{
temp = 0;
for (k=0;k<NCOLSB;k++) temp += A[i][k] * B[k][j];
AxB[i][j] = temp;
}
t3: tcomp = time_read(0) - tcomp;

/* Processors 1-(P-1) send result to proc. 0. Processor 0:ready to receive it */
if (my_node==0)
    for(i=1;i<P;++i)receive_msg(port[0][i],&AxB[i*q][0],1); /* asynchronous*/
    else send_msg(port[my_node][0],&AxB[0][0],(q*NCOLSB),1);

if (my_node==0) /* Wait for interprocessor communication to finish */
    for (i=1;i<P;++i) while (chk_dma(port[0][i]) );
else while (chk_dma(port[my_node][0])) ;

t4: tttotal = time_read(0);
/* this is including: comp + input + output + 2 timer_reads */
} /*main*/

```

DISTRIB.CMD

```

/*****
DISTRIB.CMD: Linker Command File for Distributed-Memory Program
*****/

-c                /* link using C conventions */
-stack 0x0100
-lrts40r.lib      /* get run-time support */
-lprts40r.lib
-m a.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    ROM:    org = 0x0          len = 0x1000
    RAM0:   org = 0x0002ff800  len = 0x0400    /* RAM block0 */
    RAM1:   org = 0x0002ffc00  len = 0x0400    /* RAM block1 */
    LM:     org = 0x040000000  len = 0x10000    /* local memory */
    GM:     org = 0x080000000  len = 0x20000    /* global memory */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .text:      {} > RAM0      /* code */
    .cinit:     {} > RAM1      /* initialization tables */
    .stack:     {} > RAM0      /* system stack */
    .bss :      {} > RAM1      /* global & static vars */
    .data:      {} > LM        /* for input matrix */
}

```

Appendix C: Synchronization Routine for Shared-Memory Implementation

```
*****
*
* syncount.asm : assembly language synchronization routine to provide a
* global start for all the processors. Initially, a counter in shared
* memory is set to zero. Each processor increments the counter by 1. When
* the counter equals value, the processors exit this routine. Rotating
* priority for shared-memory access should be selected. The processors
* start with a maximum cycle difference of 3 instruction cycles, which for
* practical purposes is acceptable. This routine is C-callable and uses
* registers for parameter passing.
*
* Calling conventions:
* void syncount((int *)counter,int value)          ar2 , r2
*
* where counter = synchronization counter in shared memory
*       value   = counter value to be reached.
*
*****

.global _syncount
.text
_syncount:
    LDII    *AR2,R1
    ADDI    1,R1
    CMPI    R1,R2
    STII    R1,*AR2
    BZ      L1
AGAIN:    LDI     *AR2,R1
    CMPI    R1,R2
    BNZ     AGAIN
    L1      RETS
.end
```

Appendix D: C-Callable Assembly Language Routine for Matrix Multiplication

INPUT0_A.ASM

```
*****
*
* INPUT0_A.ASM:  Contains matrix A and B input values. Matrix B is
* stored by columns.
*
*****

        .global _MAT_A
        .global _MAT_B
        .global _MAT_AxB
        .global _synch                ; counter for synchronization

        .data
_synch   .int    0
_MAT_A   .float   1.0, 2.0, 3.0, 4.0   ; stored by rows
         .float   5.0, 6.0, 7.0, 8.0
         .float   9.0, 10.0, 11.0, 12.0
         .float   13.0, 14.0, 15.0, 16.0
_MAT_B   .float   1.0, 1.0, 1.0, 1.0   ; stored by columns!!!
         .float   2.0, 2.0, 2.0, 2.0
         .float   3.0, 3.0, 3.0, 3.0
         .float   4.0, 4.0, 4.0, 4.0
_MAT_AxB .space 16                      ; must produce (stored by rows)
                                         ; 10,20,30,40
                                         ; 26,52,78,104
                                         ; 42,84,126,168
                                         ; 58,116,174,232

        .end
```

MMULT.ASM

```

*****
*
*   MMULT.ASM: Matrix multiplication (assembly language C-callable program)
*
*   mmult(&C, &A, &B, nrowa, ncolsa, ncolsb)
*       ar2, r2, r3, rc, rs, re
*
*   - Matrix A (nrowa×ncolsa): is stored by rows (row-major order)
*   - Matrix B (ncolsa×ncolsb): is stored by columns (column-major order)
*   - Matrix C (nrowa×ncolsb): is stored by rows (row-major order)
*   - "ncolsb" must be greater or equal to 2
*   - This routine uses register to pass the parameters (refer to C compiler
*     users'guide for more information)
*
*   5/1/90 : Subra Ganesan
*   10/1/90: Rosemarie Piedra
*
*****

        .global  _mmult
        .text

_mmult
        LDI      R2,AR0                ; AR0: address of A[0][0]
        LDI      R3,AR1                ; AR1: address of B[0][0]
                                           ; AR2: address of C[0][0]
        LDI      RS,IR0                ; IR0: NCOLSA
        LDI      RE,R10               ; R10: NCOLSB
        PUSH     AR5                    ; preserve registers
        PUSH     AR3
        PUSH     R5

        SUBI     1,RC,AR3              ; AR3: NROWSA-1
        SUBI     2,RS,R1              ; R1: NCOLSA-2
        SUBI     1,RE,R9              ; R9: NCOLSB-1

ROWSA
        LDI      R9,AR5

COLSB
        LDI      R2,R2                ; initialize R2
        MPYF3    *AR0++(1),*AR1++(1),R0 ; perform one multiplication
        SUBF3    R2,R2,R2             ; A(I,1)*B(1,I) -> R0
        RPTS     R1                   ; repeat the instruction NCOLSA-1 times
        MPYF3    *AR0++(1),*AR1++(1),R0
        ADDF3    R0,R2,R2

                                           ; M(I,J) * V(J) ->R0
                                           ; M(I,J-1) * V(J-1) + R2 -> R2
        DBD      AR5,COLSB            ; loop for NCOLSB times
        ADDF     R0,R2                ; last accumulate
        STF      R2,*AR2++(1)         ; result -> C[I][J]
        SUBI     IR0,AR0              ; set AR0 to point A[0][0]
        DBD      AR3,ROWSA            ; repeat NROWSA times
        ADDI     IR0,AR0              ; set AR0 to point A[1][0]
        MPYI3    IR0,R10,R5           ; R5 : NCOLSB*NROWSB(IR0)
        SUBI     R5,AR1               ; set AR1 to point B[0][0]
        POP      R5
        POP      AR3
        POP      AR5                  ; recover register values
        RETS

```

SHARED.A.C

```
/*
*****
SHARED.A.C : Parallel Matrix Multiplication (shared memory version: full
memory conflict)
-This program uses an assembly language C-callable routine for matrix
multiplication for faster program execution.
-Matrix A and C are stored by rows. Matrix B is stored by columns to
take better advantage of the assembly language implementation.
To run:
cl30 -v40 -g -o2 -as -mr shared.a.c
asm30 -v40 -s input0.a.asm
asm30 -v40 -s syncount.asm
asm30 -v40 -s mmult.asm
lnk30 mmult.obj shared.a.obj input0.a.obj shared.cmd
*****
#define NROWSA 4 /* number of rows in mat A */
#define NCOLSA 4 /* number of columns in mat A */
#define NCOLSB 4 /* number of columns in mat B */
#define P 4 /* number of processors */

extern float MAT_A[NROWSA][NCOLSA]; /* stored by rows */
extern float MAT_B[NCOLSB][NCOLSA]; /* stored by columns */
extern float MAT_AxB[NROWSA][NCOLSB]; /* stored by rows */
extern void mmult(float*C, float*A, float*B, int nrowa, int ncolsa,
int ncolsb);

extern int synch; /* synchronization for benchmarking */
extern void syncount();

float *A[NROWSA], *B[NCOLSB], *AxB[NROWSA], temp;
int *synch_p = &synch,
q = NROWSA/P,
ll = 0,
my_node, i, j, k, tcomp;

/*
*****
main()
{
asm(" OR 1800h,st"); /* cache enable */

/* accesing matrices declared in an external assembly file */
for (i=0;i<NROWSA;i++) A[i] = MAT_A[i];
for (i=0;i<NCOLSB;i++) B[i] = MAT_B[i];
for (i=0;i<NROWSA;i++) AxB[i] = MAT_AxB[i];

syncount(synch_p,P); /* global start */

if (((i = NROWSA %P) >0)) { /* load imbalancing: optional */
if (my_node<i) ++q; else ll=i;
}
ll += q*my_node; /* select beginning of row working set
for processor "my_node" */

t1: time_start(0); /* benchmarking with C40 timer */

mmult (&AxB[ll][0],&A[ll][0],&B[0][0],q,NCOLSA,NCOLSB);/* matrix mult.*/

t2 : tcomp = time_read(0); /* shared-memory benchmark */
syncount(synch_p,2*P); /* optional: if you want all processors
finish at the same time */
} /*main*/
*/

```


IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.