Lecture on programming environment

Programming language and environment for embedded multi-core processors and high performance multi-core processors

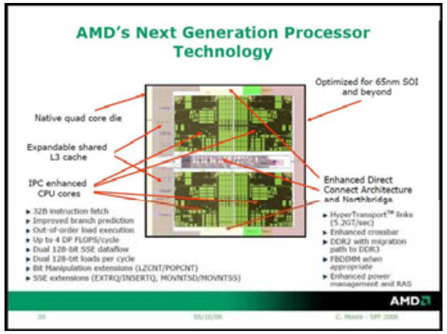
M. Sato

Contents

- Why multicore? ～ Trends of Microprocessors
  - Multi-core processor configurations: SMP vs. AMP
- How to use multicore
  - POSIX Thread
  - Communication
- Programming models
  - OpenMP
  - Cilk
  - Asynchronous RPC
- Issues and agenda

Trends of Mulitcore processors

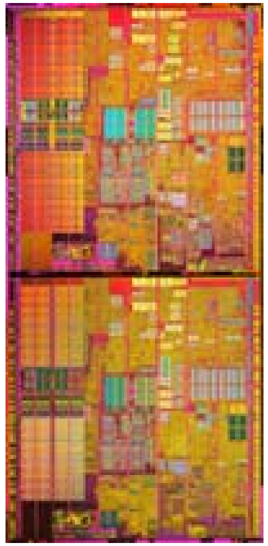
- Faster clock speed, and Finer silicon technology
  - "now clock freq is 3GHz, in future it will reach to 10GHz!?"
    - Intel changed their strategy -> multicore!
    - Clock never become faster any more
  - Silicon technology 45 nm -> 22 nm in near future!



Good news & bad news!

- Progress in Computer Architecture
  - Superpipeline, super scalar, VLIW …
  - Multi-level cache, L3 cache even in microprocessor
  - Multi-thread architecure、Intel Hyperthreading
    - Shared by multiple threads
  - Multi-core: multiple CPU core on one chip dai

Programming support is requried



Inetl ® Pentium® processor Dai of Extreme-edition

Multi-core processor：Solution of Low power by parallel processing

CPU power dissipation

$$P = N \times \alpha \times C \times V^2 \times f$$

- # CPU
- Active rate of processors
- Capacitance of circuit
- Voltage
- Clock Freq

<u>Apporach for Low power by parallel processing</u>

increase N、↑ decrease V and f, ↓ improve perf. N×f ↑

- **Decreasing V and F, makes heat dissipation and power lower within a chip**
- Progress in silicon technology 130nm ⇒ 90nm⇒65nm,22nm （Decrease **C** and **V)**
- Use a silicon process for low power (embedded processor)（Small **α**）

- Performance improvement by Multi-core（N=2～16)
  - Number of transistors are increasing by “Moore’s Law”
- Parallel processing by low power processor

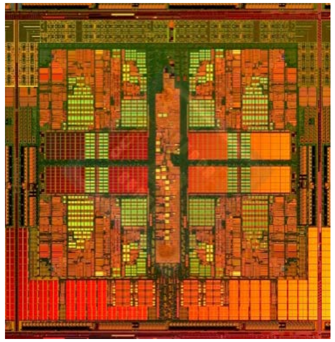
⇒ Solution by multi-core processors for High performance embedded system

Classification of Multi-core processors

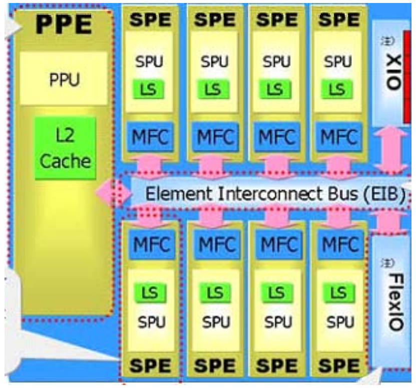
	SMP (Symmetric Multi Processor) same kinds of cores	AMP(Asymmetric Multi Processor) Different kinds of cores
Shared memory	Multi-core for sever applications ARM/NEC MPCore Renesas M32R Renesas RP1 (SH3X)	(small memory may be equipped for communication)
Distributed memory	Fujitsu FR-V(?)	IBM Cell DSP-integrated chip GPU-integrated chip

SMP and AMP

- SMP(Symmetric Multi Processor)
  - Same kind of cores integrated
  - Usually, shared memory
  - General purpose

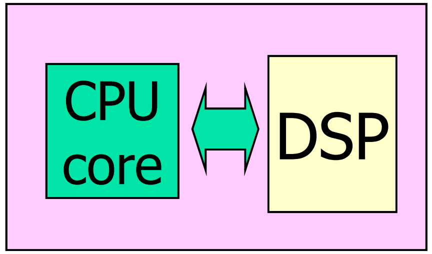


AMD quad-core

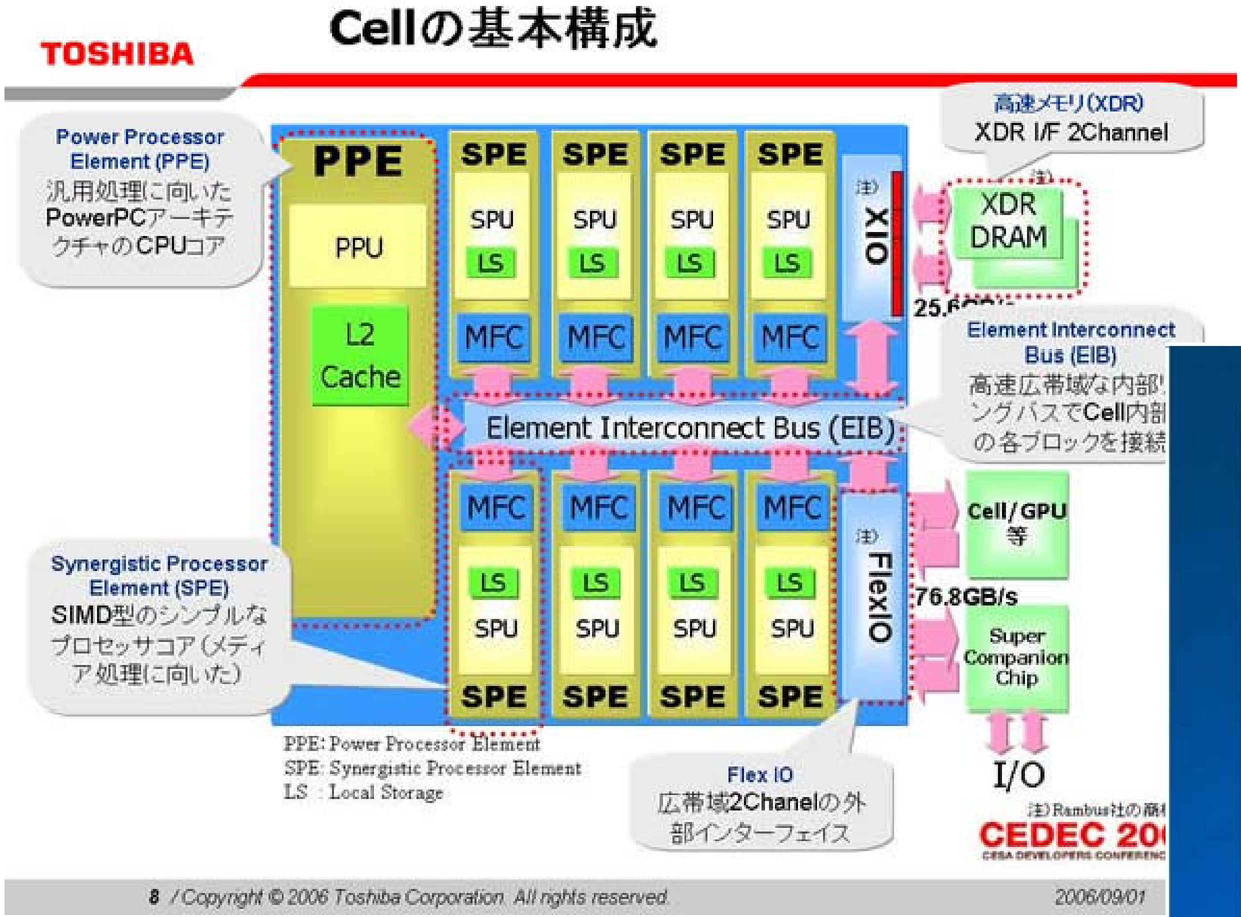


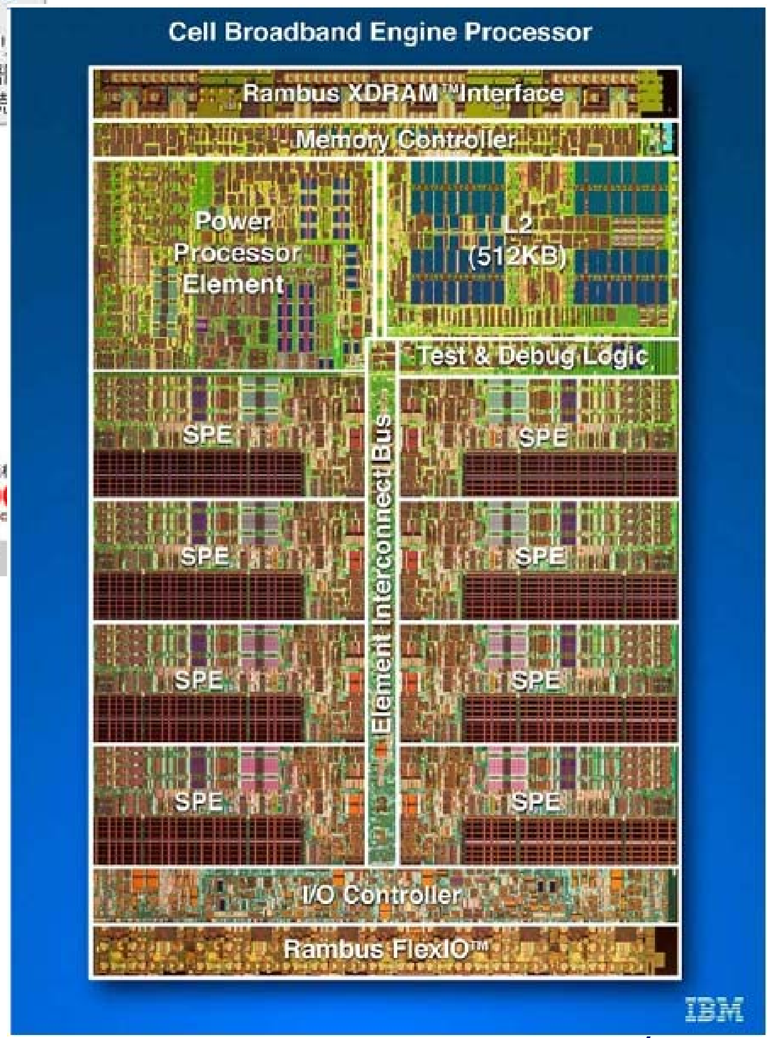
IBM Cell

- AMP(Asymmetric Multi Processor)
  - Different cores integrated, Heterogenous
  - In most case, distributed memory
  - E.g. IBM Cell processors
  - GPU-integrated, DSP-integrated
  - Special-purpose, to reduce cost



- Shared memory vs. Distributed memory
  - Important point for programming models to program multi-core processors: How to access main memory.

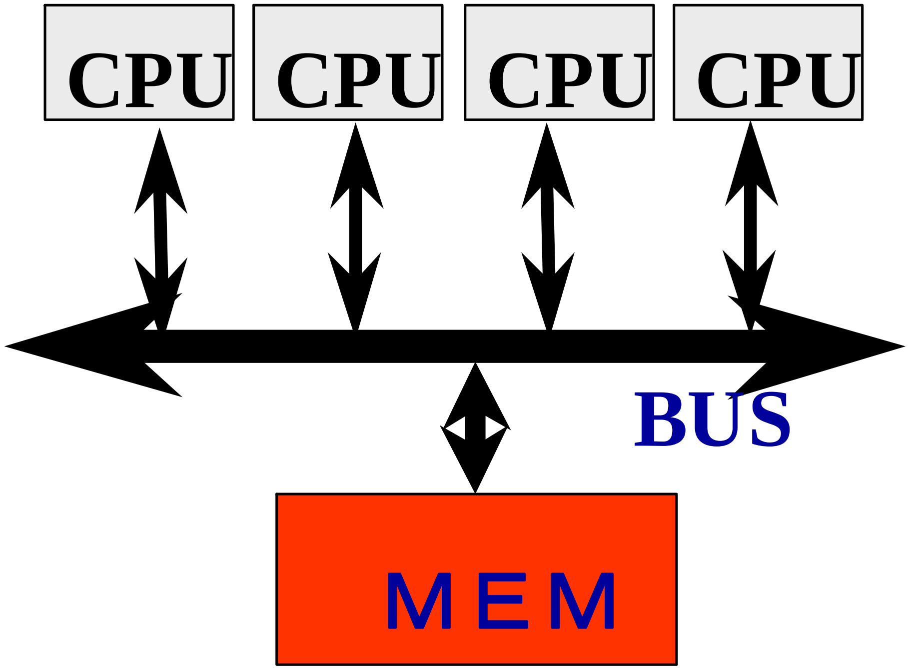




Cellのダイ

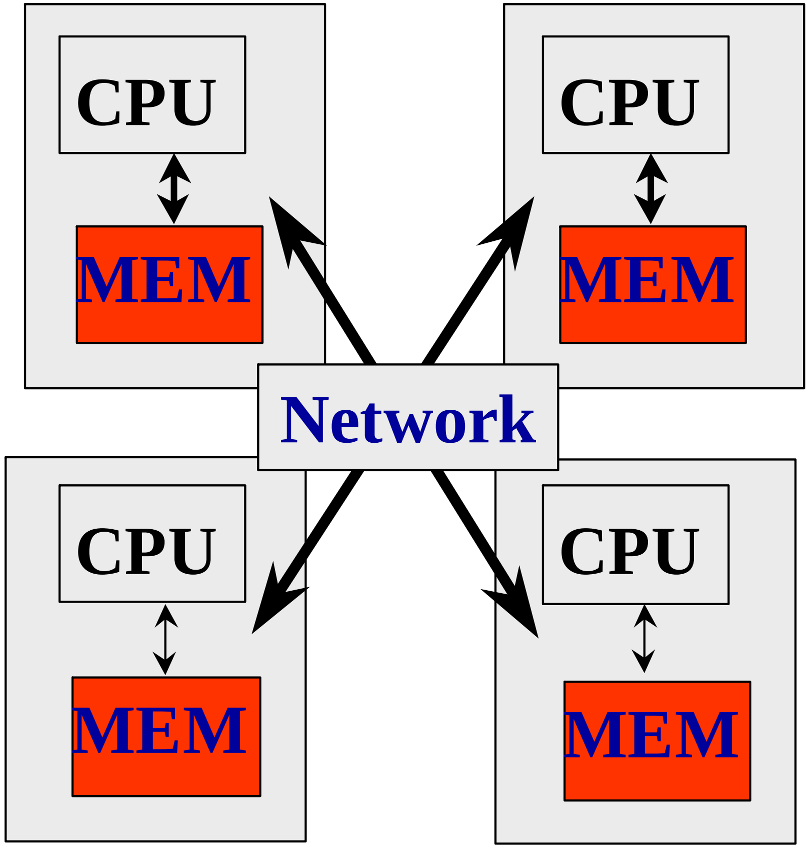
2億3400万個のトランジスタを実装するが、90nmプロセスで製造されることから、ダイ・サイズは221mm2と意外と小さい。細長い部分がSPUである。

Shared memory multi-processor system



- **Multiple CPUs share main memory**
- **Threads executed in each core(CPU) communicate with each other by accessing shared data in main memory.**
- **Enterprise Server**
- **SMP Multi-core processors**

Distributed memory multi-processor

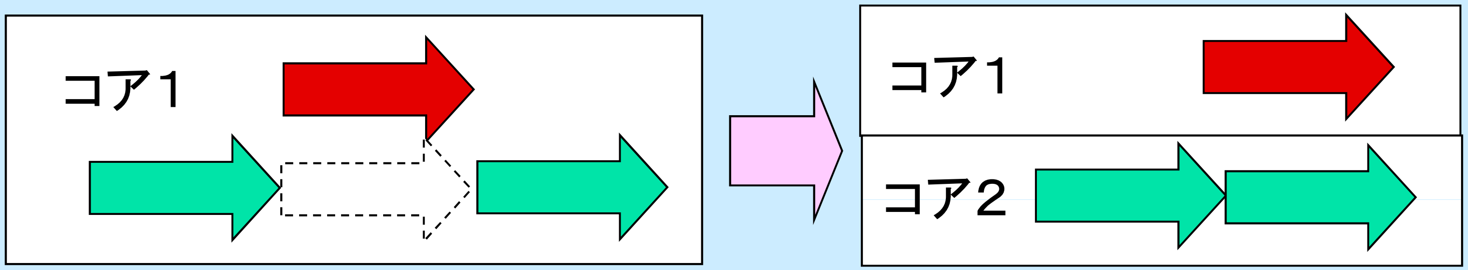


- ◆**System with several computer of CPU and memory, connected by network.**
- ◆**Thread executed in each computer communicate with each other by exchanging data (message) via network.タ**
- ◆**PC Cluster**
- ◆**AMP Multi-core processor**

How to use Multi-core processor (1)

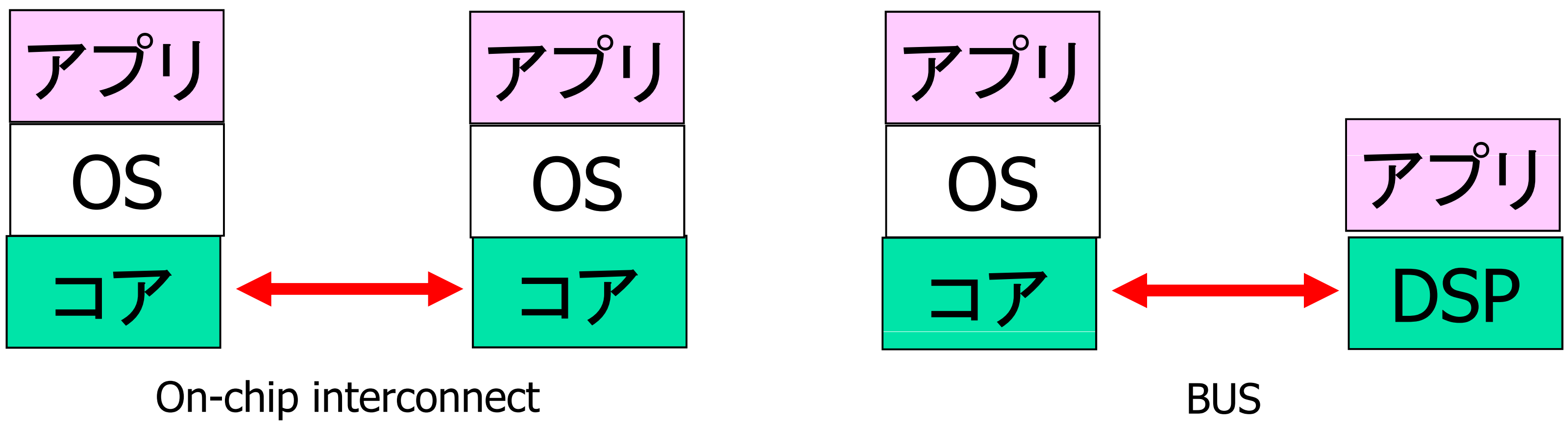
- Run process or threads on each core
  - Possible mainly on shared memory SMP multi-core processors
  - Most embedded applications are a multi-task (multi-process) program.
  - It may require any particular modification.

- In some cases, multi-task program running on a single core cannot be executed in multi-core (SMP)
  - The case of using high-priority non-preemptive execution of Real time OS as an implementation of critical section.
  - In multi-core environment, high-priority execution does not mean “non-preemptive” execution since other thread run in parallel physically.
  - Use lock properly to implement critical section.



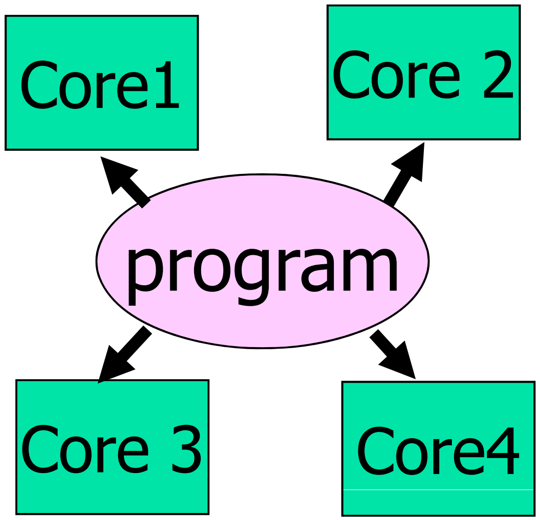
How to use Multi-core processor (2)

- Use each core for different functions (hetero)
  - A typical usage of AMP multi-core processor
    - It can be applied for SMP-type multi-core , without shared memory.
    - So far, this kind of applications use several kinds of different chips.
  - In this case, individual OS can run on each core. (DSP has no OS)
    - May use different kind of OS, e.g. Linux and RTOS
  - Communication by using on-chip interconnect or bus
    - Can use RPC model for programming

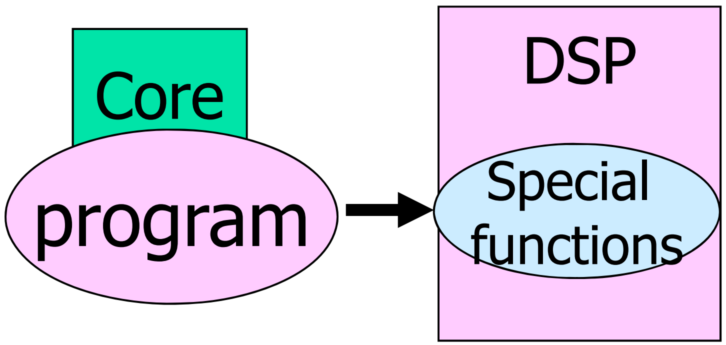


How to use Multi-core processor (3)

- For high performance (common goal?)
  - Parallel processing by multiple cores
  - Can use OpenMP for shared memory SMP
    - Technologies which is used in high-end platform
  - Use DSP or GPU to accelerate computation in AMP

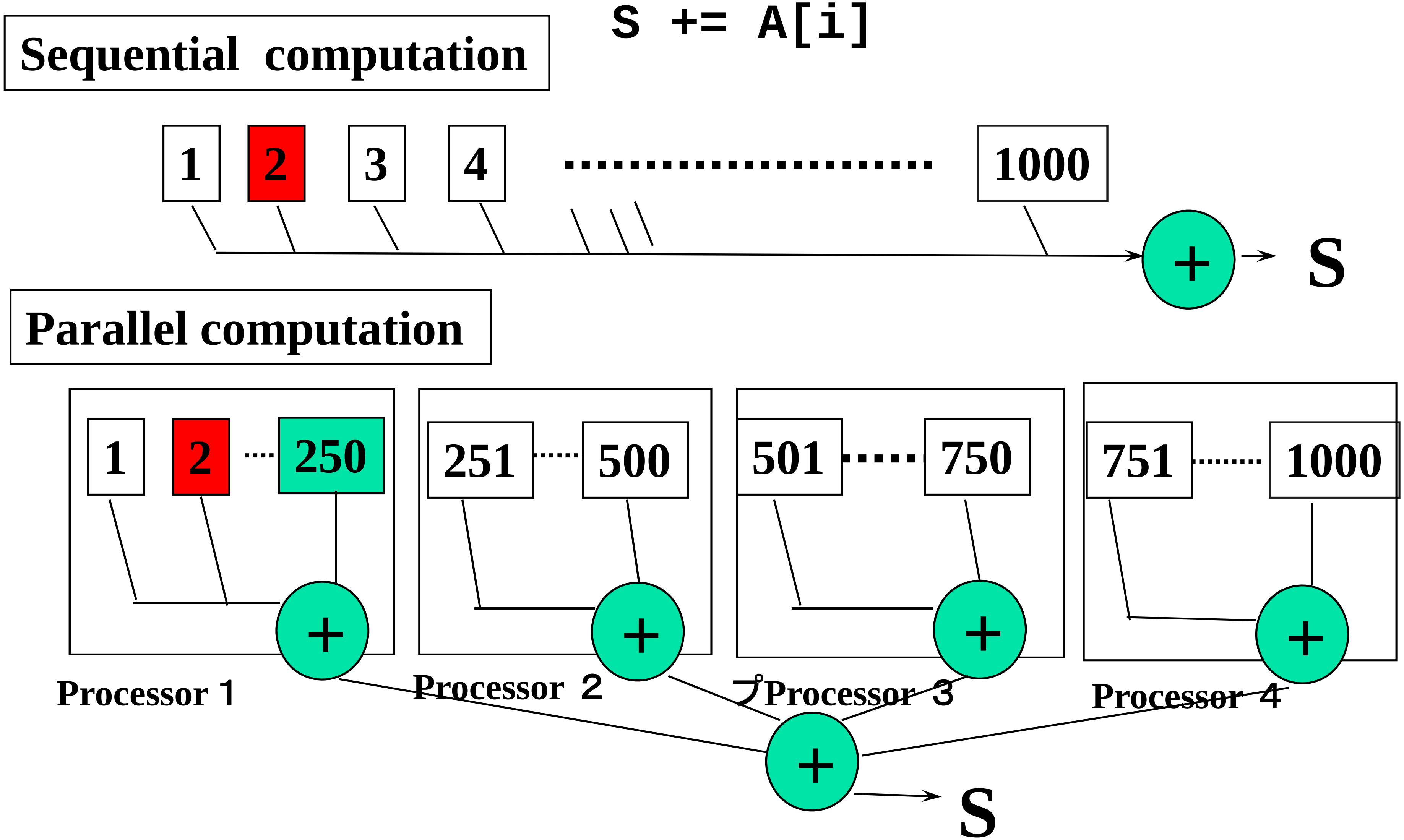


Work-sharing by multiple cores



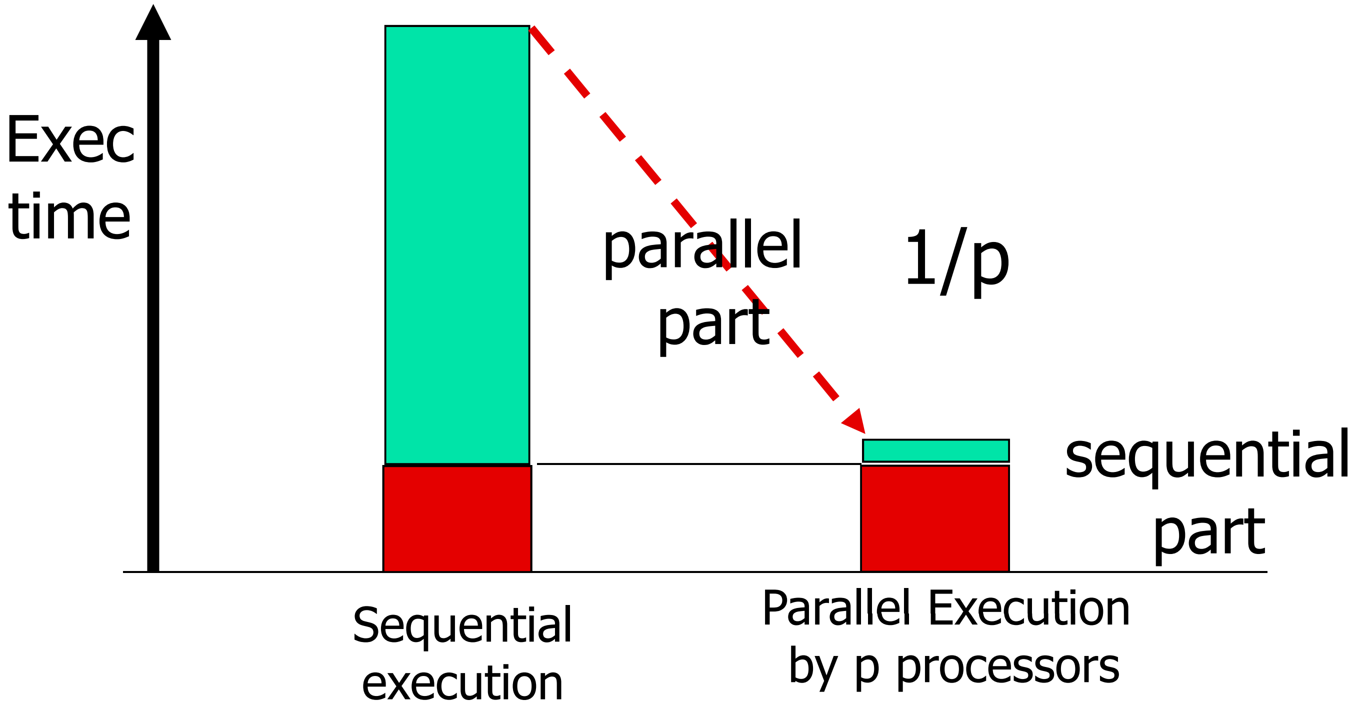
Very simple example of parallel computing for high performance

```
for(i=0;i<1000; i++)
    S += A[i]
```



Speedup by parallel computing："Amdahl's low"

- Amdahl's low
  - Suppose execution time of sequential part T_1, ratio of sequential part α, execution time by parallel computing using p processors T_p is (no more than) $T_p = \alpha * T_1 + (1-\alpha) * T_1/p$
  - Since some part must be executed sequentially, speedup is limited by the sequential part.



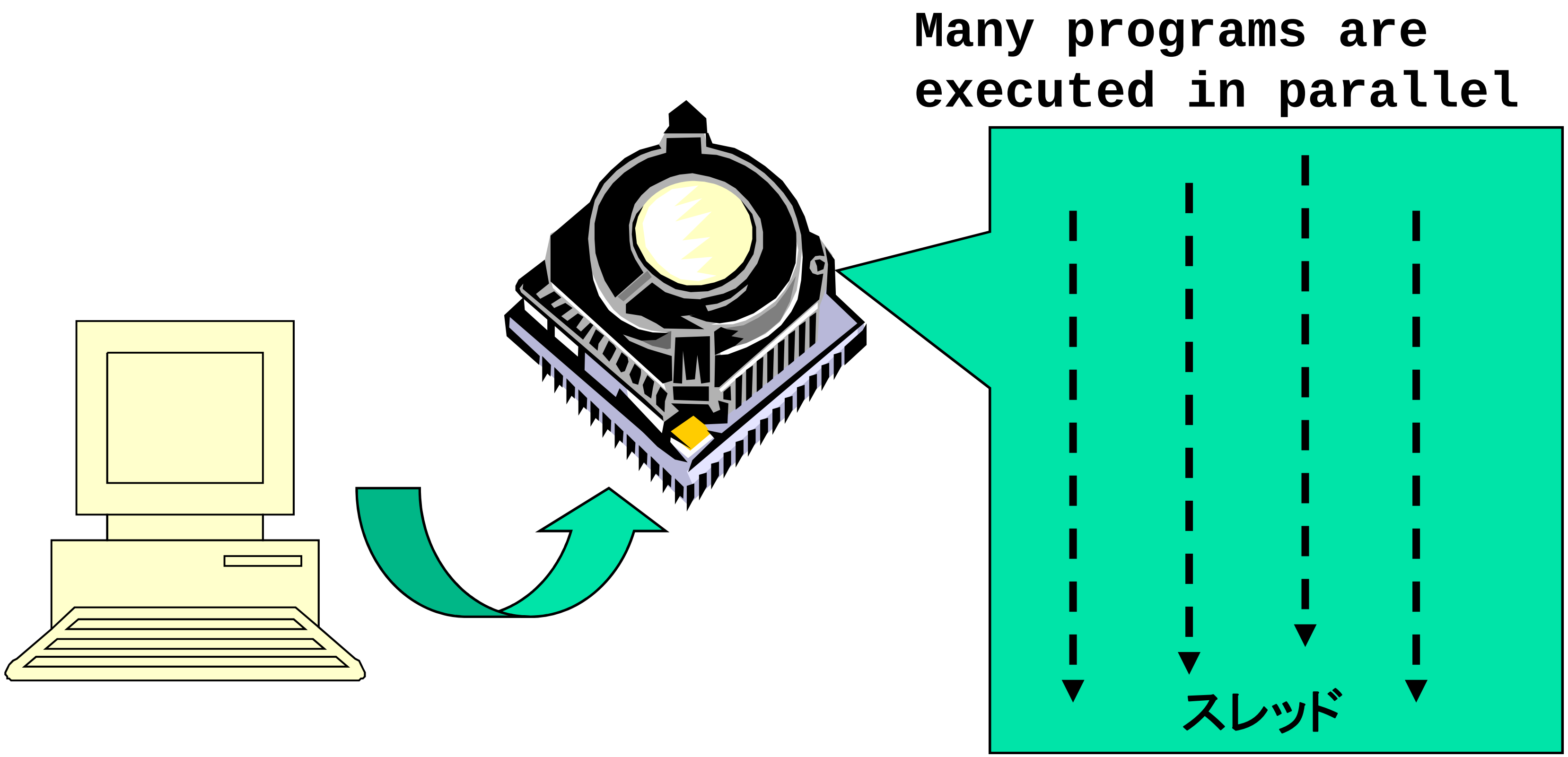
Parallel programming model

- Message passing programming model
  - Parallel programming by exchange data (message) between processors (nodes)
  - Mainly for distributed memory system (possible also for shared memory)
  - Program must control the data transfer explicitly.
  - Programming is sometimes difficult and time-consuming
  - Program may be scalable (when increasing number of Proc)

- Shared memory programming model
  - Parallel programming by accessing shared data in memory.
  - Mainly for shared memory system. (can be supported by software distributed shared memory)
  - System moves shared data between nodes (by sharing)
  - Easy to program, based on sequential version
  - Scalability is limited. Medium scale multiprocessors.

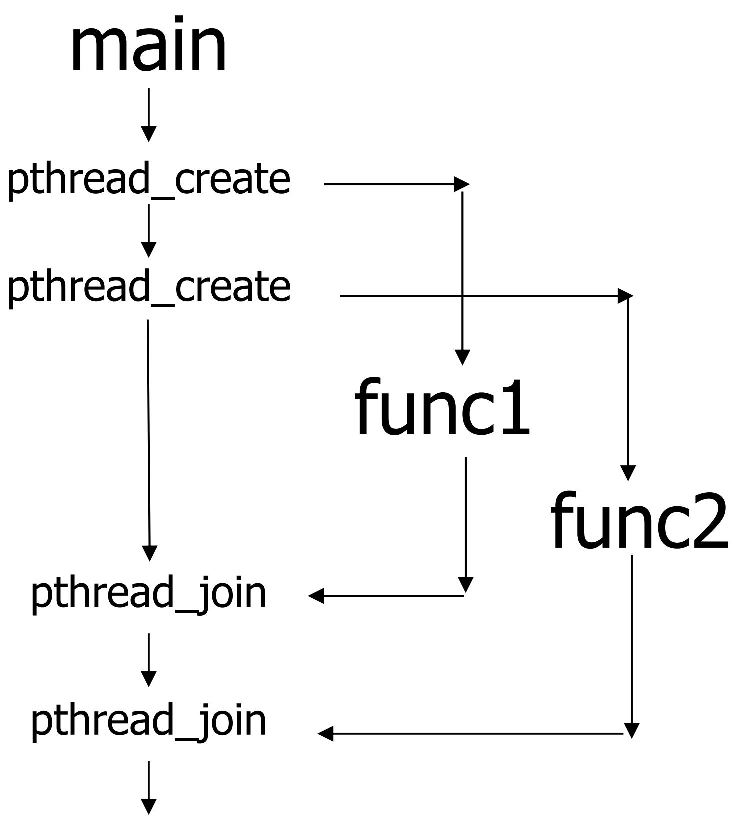
Multithread(ed) programming

- Basic model for shared memory
- Thread of execution = abstraction of execution in processors.
  - Different from process
    - Procss = thread + memory space
  - POSIX thread library = pthread



POSIX thread library

- Create thread: thread_create
- Join threads: pthread_join
- Synchronization, lock




```c
#include <pthread.h>

void func1( int x );   void func2( int x );

main() {
        pthread_t t1 ;
        pthread_t t2 ;
           pthread_create( &t1, NULL,
                           (void *)func1, (void *)1 );
           pthread_create( &t2, NULL,
                           (void *)func2, (void *)2 );
           printf("main()¥n");
           pthread_join( t1, NULL );
           pthread_join( t2, NULL );
}
void func1( int x ) {
       int i ;
        for( i = 0 ; i<3 ; i++ ) {
             printf("func1( %d ): %d ¥n",x, i );
         }
}
void func2( int x ) {
             printf("func2( %d ): %d ¥n",x);
}
```

Programming using POSIX thread

- Create threads

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
      pthread_join();
```

Thread =
Execution of program

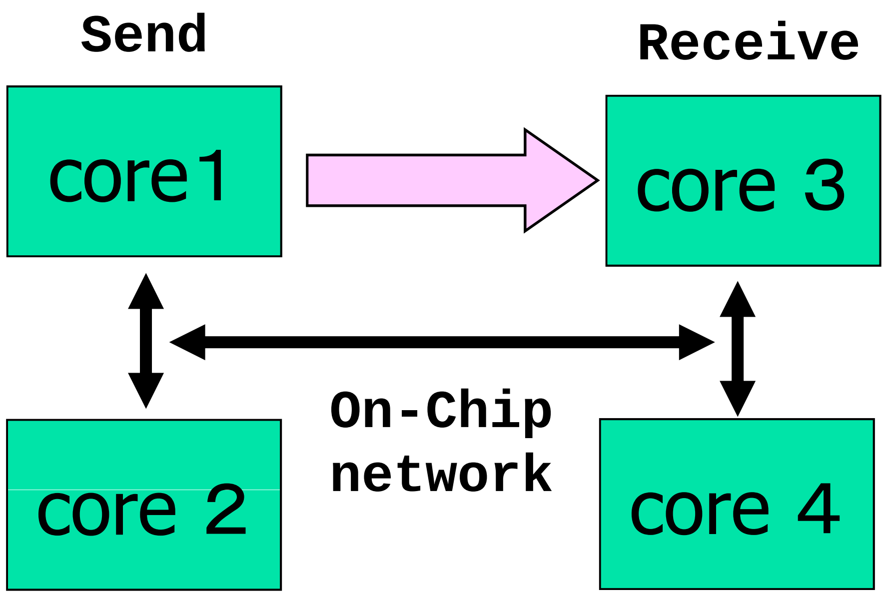
- Divide and assign iterations of loop
- Synchronization for sum

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

Message passing programming

- General programming paradigm for distributed memory system.
  - Data exchange by "send" and "receive"

- Communication library, layer
  - POSIX IPC, socket
  - TIPC (Transparent Interprocess Communication)
  - LINX (on Enea's OSE Operating System)
  - MCAPI (Multicore Communication API)
  - MPI (Message Passing Interface)



Simple example of Message Passing Programming

- Sum up 1000 element in array

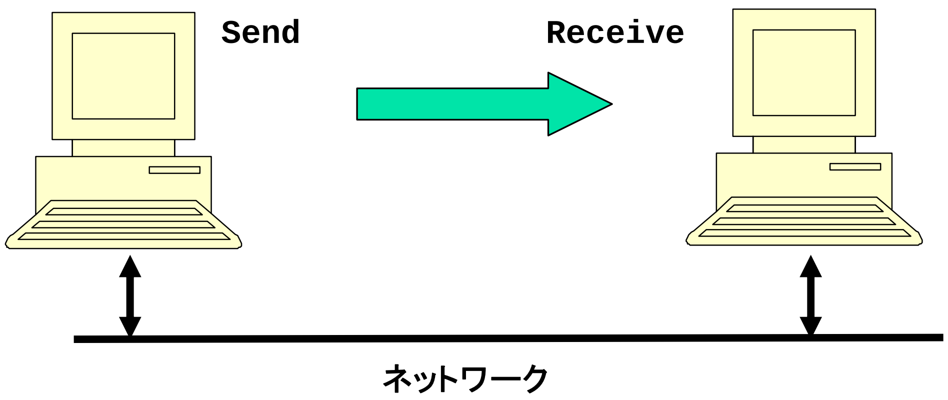
```
int a[250]; /* 250 elements are allocated in each node */

main(){    /* start main in each node */
   int i,s,ss;
   s=0;
   for(i=0; i<250;i++) s+= a[i]; /*compute local sum*/
   if(myid == 0){   /* if processor 0 */
      for(proc=1;proc<4; proc++){
         recv(&ss,proc);  /* receive data from others*/
         s+=ss;   /*add local sum to sum*/
      }
   } else {  /* if processor 1,2,3 */
      send(s,0);  /* send local sum to processor 0 */
   }
}
```

Parallel programming using MPI

- MPI (Message Passing Interface)
- Mainly, for High performance scientific computing
- Standard library for message passing parallel programming in high-end distributed memory systems.
  - Required in case of system with more than 100 nodes.
  - Not easy and time-consuming work
    - "assembly programming" in distributed programming

Over-specs for Embedded system Programming?!

- Communication with message
  - Send/Receive
- Collective operations
  - Reduce/Bcast
  - Gather/Scatter



Programming in MPI

```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```

Programming in MPI

```
sum = 0.0;
for (i = 0; i < 1000/N_PE; i++){
   sum+ = A[i];
}

if(myid == 0){
   for(i = 1; i < numprocs; i++){
       MPI Recv(&t,1,MPI DOUBLE,i,MY TAG,MPI COMM WORLD,&status
        sum += t;
     }
} else
        MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
/* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_
 MPI_Barrier(MPI_COMM_WORLD);
...
MPI_Finalize();
return 0;
}
```

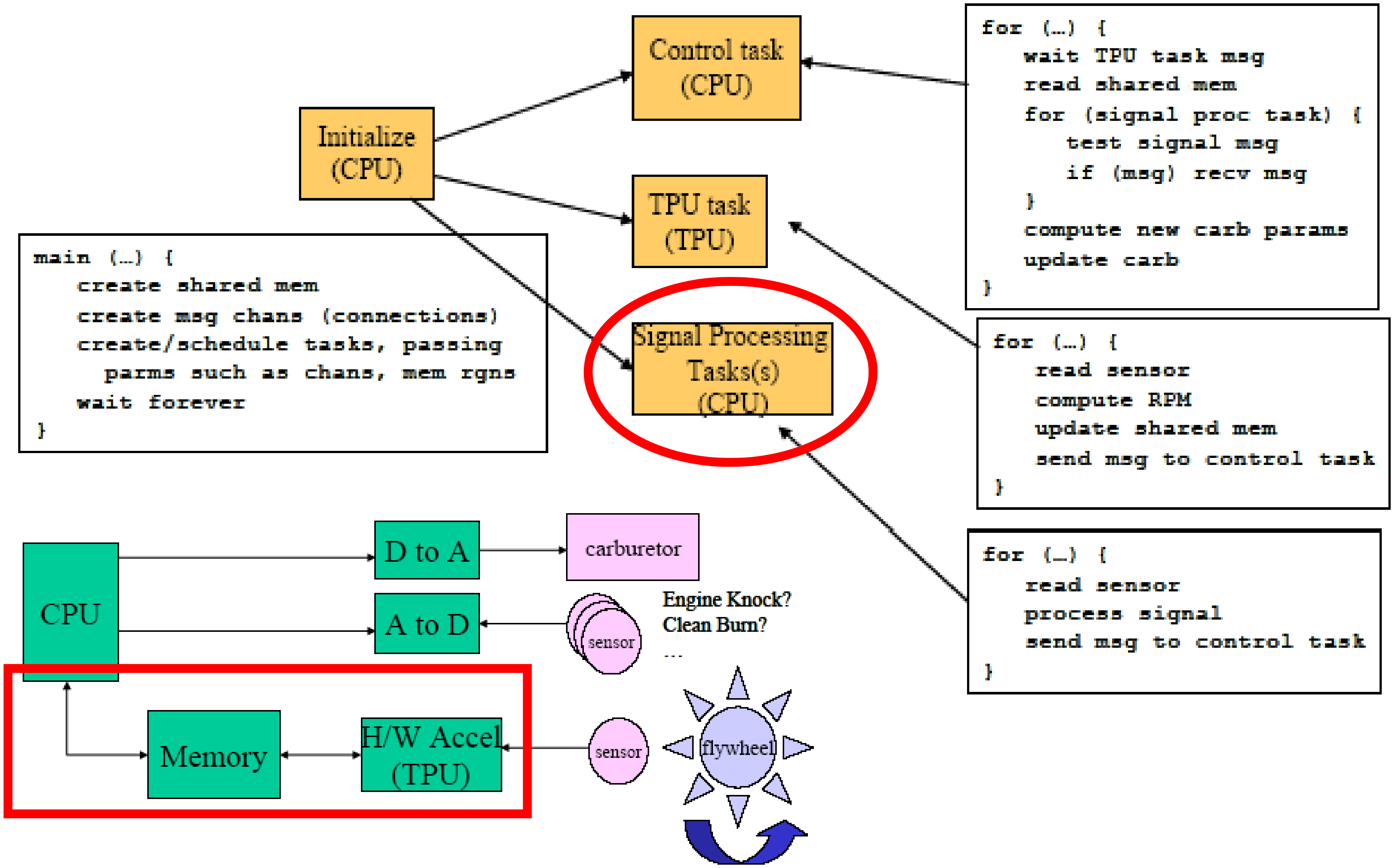
MCAPI

- MCAPI (Multicore Communication API)
  - Communication API defined by Multicore Association (www.multicore-association.org, Intel, Freescale, TI, NEC)
  - V1.063 at March 31, 2008
  - Using with MRAPI (Resource Management API)
  - Easy than MPI、hetero, scalable, fault tolerance(?), general

- 3 Basic functions
  - 1. Messages – connection-less datagrams.
  - 2. Packet channels – connection-oriented, uni-directional, FIFO packet streams.
  - 3. Scalar channels – connection-oriented single word uni-directional, FIFO packet streams.

- MCAPI's objective is to provide a limited number of calls with sufficient communication functionality while keeping it simple enough to allow efficient implementations.

Example


Control task
(CPU)
Initialize
(CPU)
TPU task
(TPU)
Signal Processing
Tasks(s)
(CPU)
for (…) {
wait TPU task msg
read shared mem
for (signal proc task) {
test signal msg
if (msg) recv msg
}
compute new carb params
update carb
}
main (…) {
create shared mem
create msg chans (connections)
create/schedule tasks, passing
parms such as chans, mem rgns
wait forever
}
for (…) {
read sensor
compute RPM
update shared mem
send msg to control task
}
for (…) {
read sensor
process signal
send msg to control task
}
CPU
D to A
A to D
carburetor
sensor
Engine Knock?
Clean Burn?
…
Memory
H/W Accel
(TPU)
sensor
flywheel

```
////////////////////////////////////////////
// The TPU task
////////////////////////////////////////////
void TPU_Task() {
	char* sMem;
	size_t msgSize;
	mcapi_endpoint_t cntrl_endpt, cntrl_remote_endpt;
	mcapi_sclchan_send_hndl_t cntrl_chan;
	mcapi_request_t r1;
	mcapi_status_t err;

	// init the system
	mcapi_initialize(TPU_NODE, &err);
	CHECK_STATUS(err);
	cntrl_endpt =
	mcapi_create_endpoint(TPU_PORT_CNTRL, &err);
	CHECK_STATUS(err);
	mcapi_get_endpoint_i(CNTRL_NODE,
	CNTRL_PORT_TPU,
	&cntrl_remote_endpoint, &r1, &err);
	CHECK_STATUS(err);

	// wait on the remote endpoint
	mcapi_wait(&r1,NULL,&err);
	CHECK_STATUS(err);

	// now get the shared mem ptr
	mcapi_msg_recv(cntrl_endpt, &sMem,
	sizeof(sMem), &msgSize, &err);
	CHECK_MEM(sMem);
	CHECK_STATUS(err);

	// NOTE – connection handled by control task
	// open the channel
	mcapi_open_sclchan_send_i(&cntrl_chan,
	cntrl_endpt, &r1, &err);
	CHECK_STATUS(err);
	// wait on the open
	mcapi_wait(&r1,NULL,&err);
	CHECK_STATUS(err);

	// ALL bootstrapping is finished, begin  processing
	while (1) {
		// do something that updates shared mem
		sMem[0] = 1;
		// send a scalar flag to cntrl process
		// indicating sMem has been updated
		mcapi_sclchan_send_uint8(cntrl_chan,
		(uint8_t) 1,&err);
		CHECK_STATUS(err);
	}
}
```

What's OpenMP?

- Programming model and API for shared memory parallel programming
  - It is not a brand-new language.
  - Base-languages(Fortran/C/C++) are extended for parallel programming by directives.
  - Main target area is scientific application.
  - Getting popular as a programming model for shared memory processors as multi-processor and multi-core processor appears.

- OpenMP Architecture Review Board (ARB) decides spec.
  - Initial members were from ISV compiler venders in US.
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - Latest version, OpenMP 3.0

- http://www.openmp.org/

Programming using POSIX thread

- Create threads

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
       pthread_join();
```

Thread =
Execution of program

- Divide and assign iterations of loop
- Synchronization for sum

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

Programming in OpenMP

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```

OpenMP API

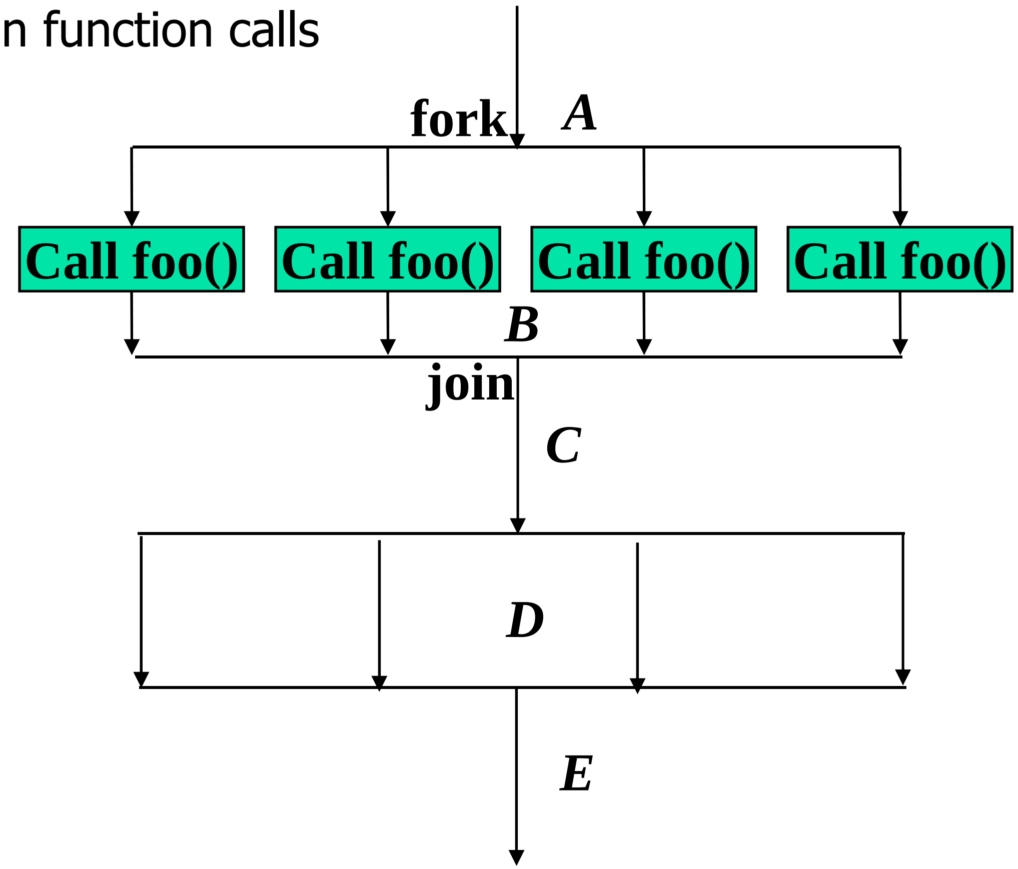
- It is not a new language!
  - Base languages are extended by compiler directives/pragma, runtime library, environment variable.
  - Base languages：Fortran 90, C, C++
    - Fortran： directive line starting with !$OMP
    - C: directive by #pragma omp

- Different from automatic parallelization
  - OpenMP parallel execution model is defined explicitly by a programmer.

- If directives are ignored (removed), the OpenMP program can be executed as a sequential program
  - Can be parallelized in incrementally
  - Practical approach with respect to program development and debugging.
  - Can be maintained as a same source program for both sequential and parallel version.

OpenMP Execution model

- Start from sequential execution
- Fork-join Model
- parallel region
  - Duplicated execution even in function calls




```
… A …
#pragma omp parallel
{
    foo(); /* ..B... */
}
… C ….
#pragma omp parallel
{
… D …
}
… E …
```

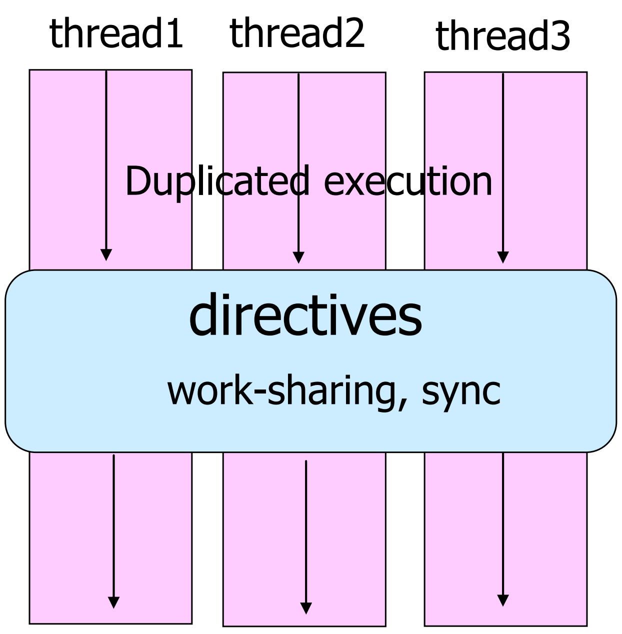
Parallel Region

- A code region executed in parallel by multiple threads (team)
  - Specified by Parallel constructs
  - A set of threads executing the same parallel region is called "team"
  - Threads in team execute the same code in region (duplicated execution)

```
#pragma omp parallel
{
    ...
    ... Parallel region...
    ...
}
```

Work sharing Constructs

- Specify how to share the execution within a team
  - Used in parallel region
  - `for` Construct
    - Assign iterations for each threads
    - For data parallel program
  - `Sections` Construct
    - Execute each section by different threads
    - For task-parallelism
  - `Single` Construct
    - Execute statements by only one thread
  - Combined Construct with parallel directive
    - `parallel for` Construct
    - `parallel sections` Construct



For Construct

- Execute iterations specified For-loop in parallel
- For-loop specified by the directive must be in *canonical shape*

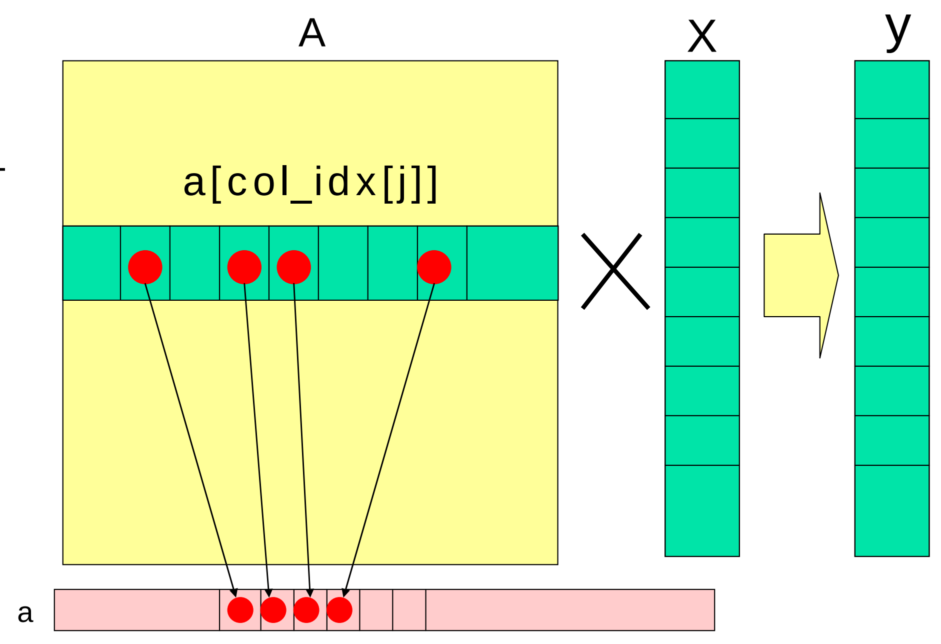
```
#pragma omp for [clause…]
 for(var=lb; var logical-op ub; incr-expr)
    body
```

- *Var* must be loop variable of integer or pointer(automatically private)
- *incr-expr*
  - ++*var*, *var*++, - -*var*, *var*- -, *var*+=*incr*, *var*-=*incr*
- *logical-op*
  - <、<=、>、>=
- Jump to ouside loop or break are not allows
- Scheduling method and data attributes are specified in *clause*

Example code

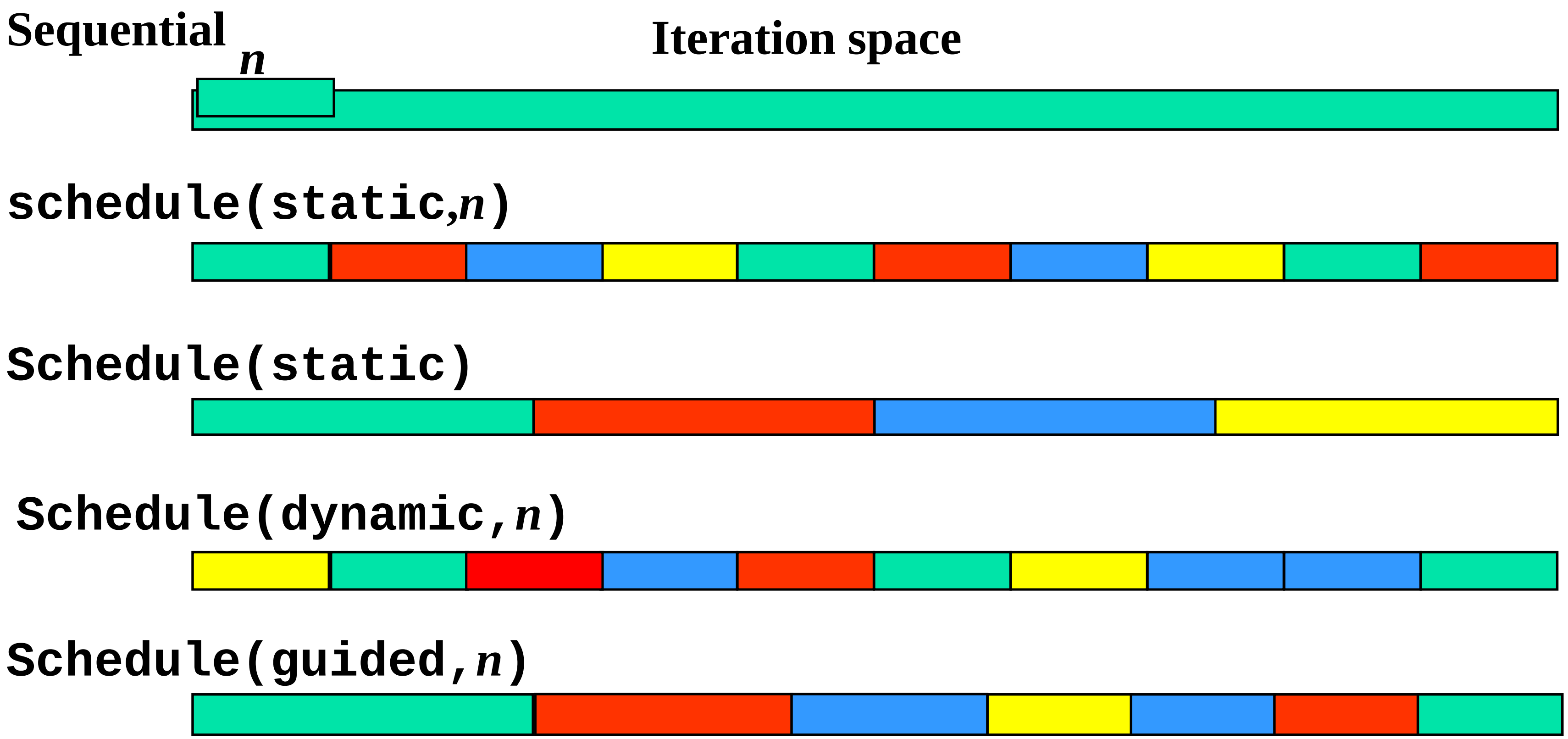
Sparse matrix vector product

```
Matvec(double a[],int row_start,int col_idx[],
 double x[],double y[],int n)
{
   int i,j,start,end; double t;
#pragma omp parallel for private(j,t,start,end)
   for(i=0; i<n;i++){
      start=row_start[i];
      end=row_start[i+1];
      t = 0.0;
      for(j=start;j<end;j++)
         t += a[j]*x[col_idx[j]];
      y[i]=t;
  }
}
```



Scheduling methods of parallel loop

- #processor = 4



Data scope attribute clause

- Clause specified with `parallel`consruct、work sharing construct
- `shared(var_list)`
  - Specified variables are shared among threads.
- `private(var_list)`
  - Specified variables replicated as a private variable
- `firstprivate(var_list)`
  - Same as private, but initialized by value before loop.
- `lastprivate(var_list)`
  - Same as private, but the value after loop is updated by the value of the last iteration.
- `reduction(op:var_list)`
  - Specify the value of variables computed by reduction operation op.
  - Private during execution of loop, and updated at the end of loop

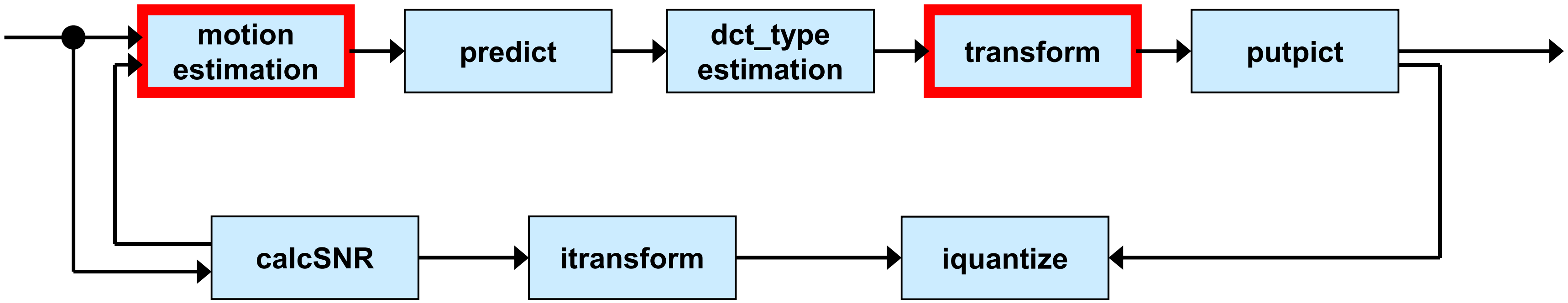
Barrier directive

- Sync team by barrier synchronization
  - Wait until all threads in the team reached to the barrier point.
  - Memory write operation to shared memory is completed (flush) at the barrier point.
  - Implicit barrier operation is performed at the end of parallel region, work sharing construct without `nowait` clause

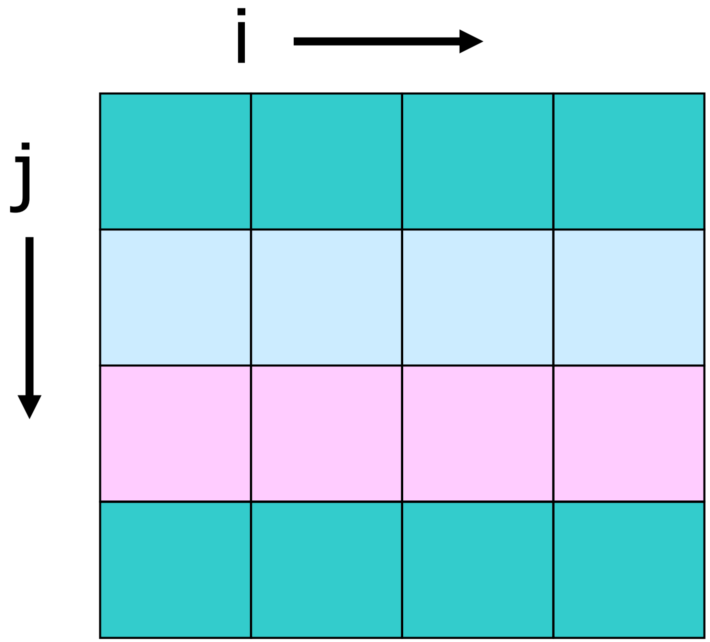
```
#pragma omp barrier
```

MediaBench

- MPEG2 encoder by OpenMP.

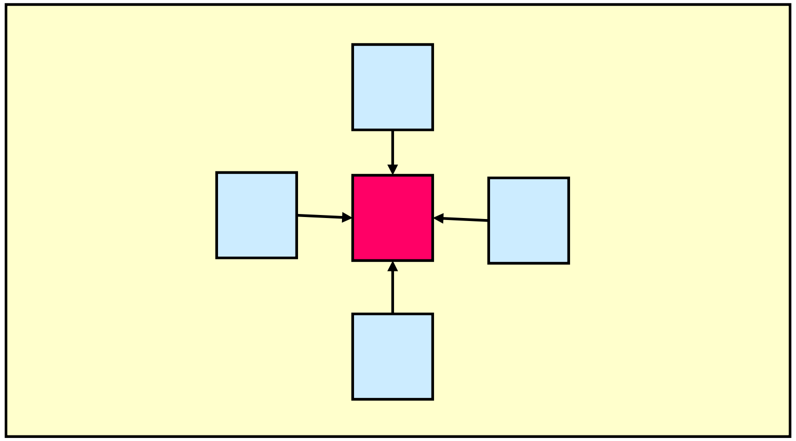



```
/*loop through all macro-blocks of the picture*/
#pragma omp parallel private(i,j,myk)
{
#pragma omp for
     for (j=0; j<height2; j+=16)
     {
          for (i=0; i<width; i+=16)
          {
              ... loop body ...
          }
     }
}
```



Example of OpenMP program：laplace

- Explicit solver of Laplace equation
  - Stencil operation: update value with 4-points of up/down/left/right.
  - Use array of “old” and “new”. Compute new by old and replace old with new.
  - Typical parallelization by domain decomposition
  - At each iteration, compute residual



- OpenMP version: lap.c
  - Parallelize 3 loops
    - OpenMP support only loop parallelization of outer loop.
  - For loop directive is orphan, in dynamic extent of parallel directive.

```
void lap_solve()
{
    int x,y,k;
    double sum;

#pragma omp parallel private(k,x,y)
{
    for(k = 0; k < NITER; k++){
         /* old <- new */
#pragma omp for
         for(x = 1; x <= XSIZE; x++)
           for(y = 1; y <= YSIZE; y++)
             uu[x][y] = u[x][y];
         /* update */
#pragma omp for
         for(x = 1; x <= XSIZE; x++)
           for(y = 1; y <= YSIZE; y++)
             u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
    }
 }

/* check sum */
    sum = 0.0;
#pragma omp parallel for private(y) reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
         for(y = 1; y <= YSIZE; y++)
           sum += (uu[x][y]-u[x][y]);
    printf("sum = %g¥n",sum);
}
```

Update in OpenMP3.0

- The concept of "task" is introduced:
  - An entity of thread created by Parallel construct and Task construct.
  - Task Construct & Taskwait construct
- Interpretation of shared memory consistency in OpenMP
  - Definition of Flush semantics
- Nested loop
  - Collapse clauses
- Specify stack size of thread.
- constructor, destructor of private variables in C++

Example of Task Constructs

```
struct node {
        struct node *left;
        struct node *right;
};

void postorder_traverse( struct node *p ) {
        if (p->left)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->left);
        if (p->right)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->right);
        #pragma omp taskwait
        process(p);
}
```

What about performance?

- OpenMP really speedup my problem?!
- It depends on hardware and problem size/characteristics
- Esp. problem sizes is an very important factor
  - Trade off between overhead of parallelization and grain size of parallel execution.
- To understand performance, …
  - How to lock
  - How to exploit cache
  - Memory bandwidth

Some experience of OpenMP performance of embedded multicore processors

- SMP multicore for embedded
  - renesas： M32700
  - ARM+NEC： MPCore
  - Hitachi+renesas： RP1
- For comparison
  - Intel: Core2Quad Q6600 (Desktop/server)

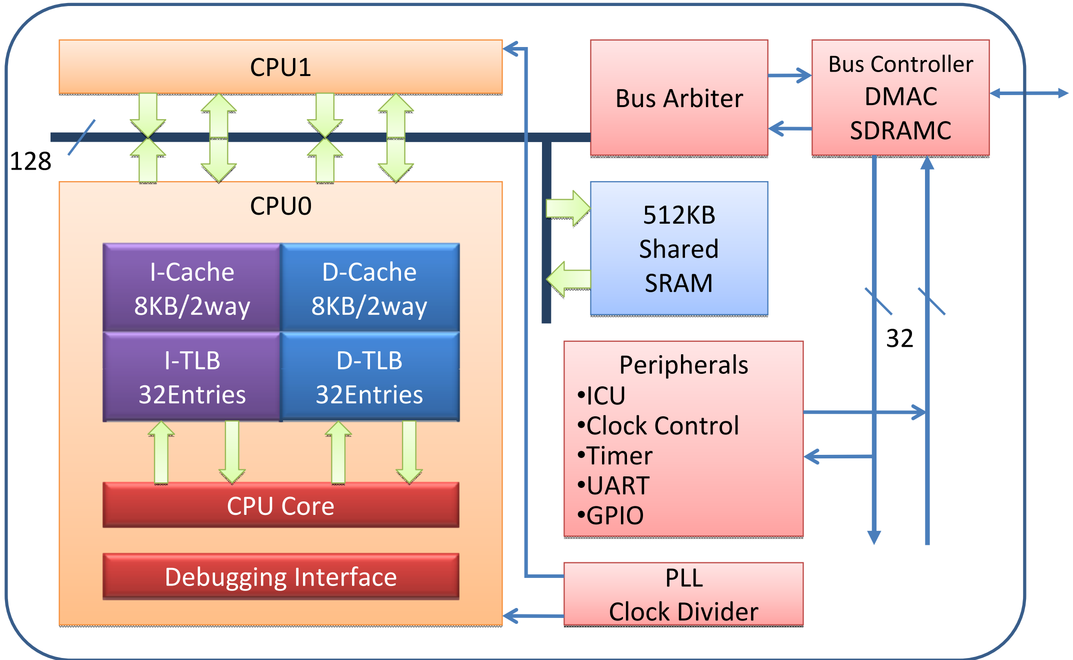
塙 敏博,李 珍泌,今田 貴之,木村 英明[2] 佐藤 三久[1,],朴 泰祐
"OpenMP を用いた並列ベンチマークプログラムによる
組込み向けマルチコアプロセッサの評価", SWoPP 2008

Renesas M32R (M32700)

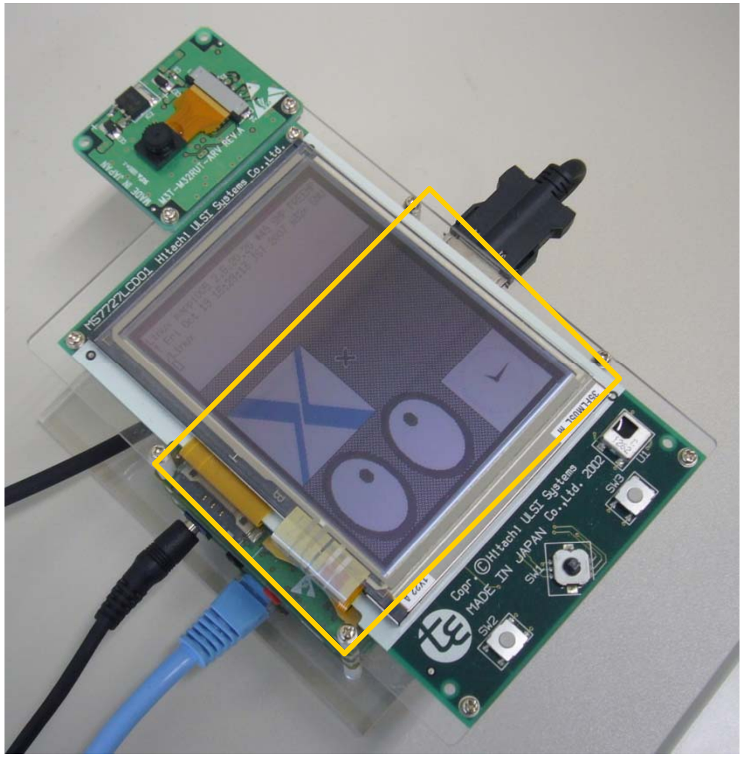
- M32R-II core x 2
  - 7-stage pipeline
  - 32bit instruction（1命令同時発行+16bit命令(2命令同時発行可能)
  - No floating unit
    - gcc付属の浮動小数点ライブラリ (soft-float)

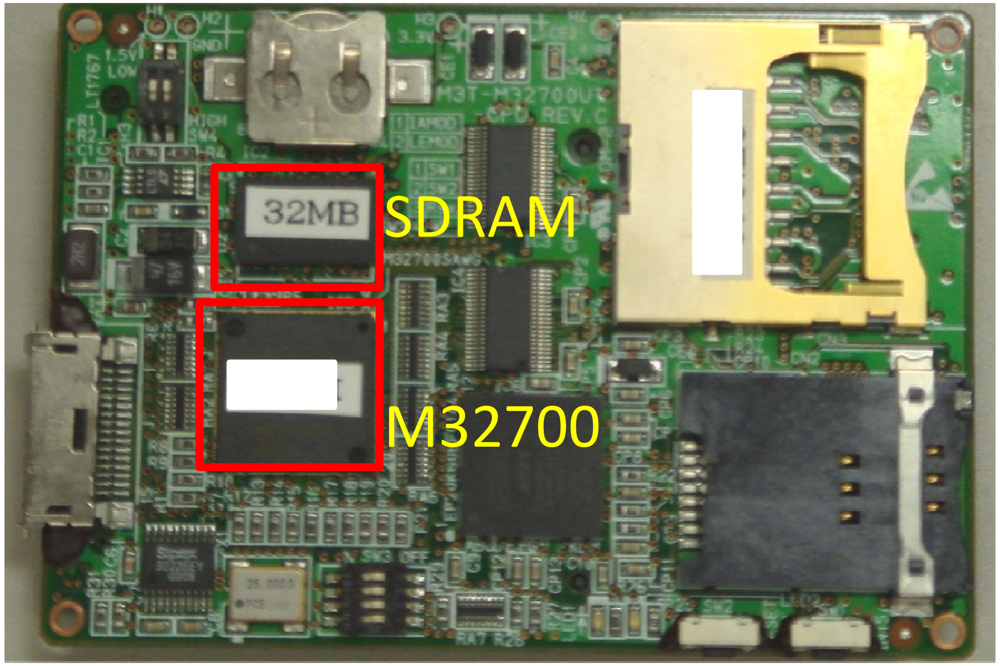
- On-chip 512KB SRAM
  - Not used in our experiment
- SDRAM controller

- μT-Engine
  M3T-32700UTを使用



M32700 development kit



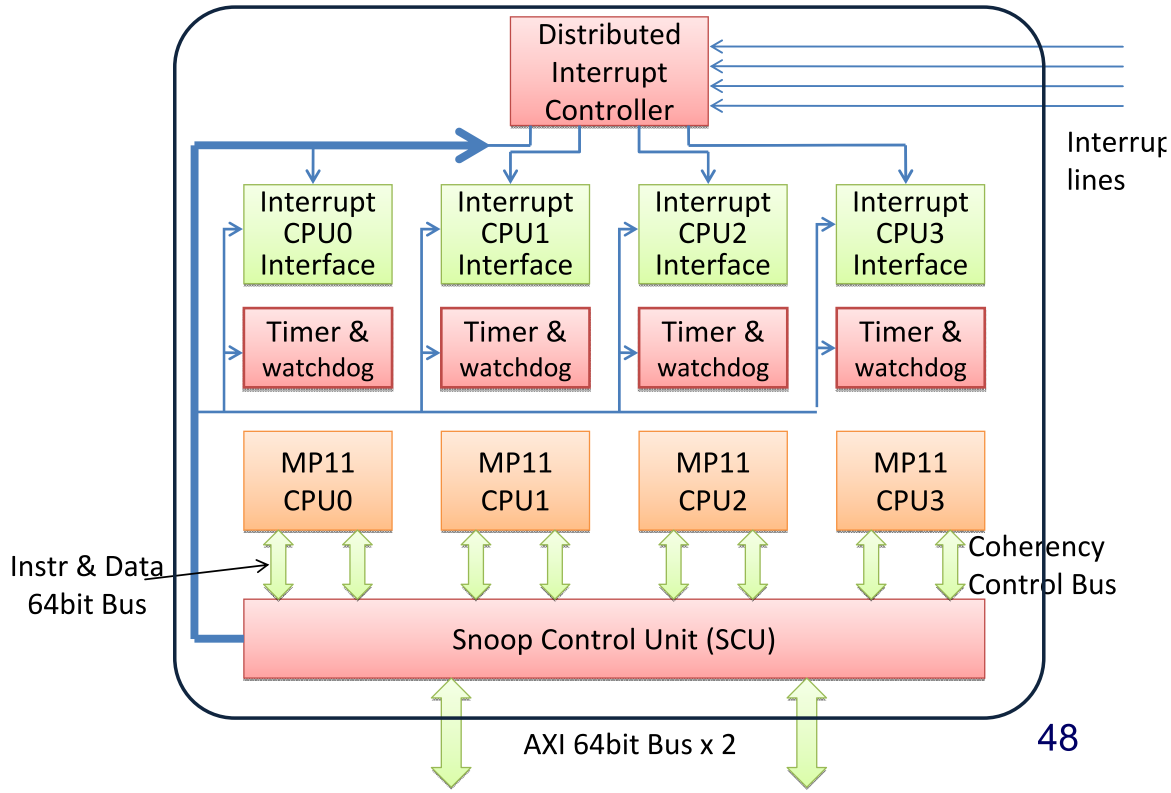


ARM MPCore

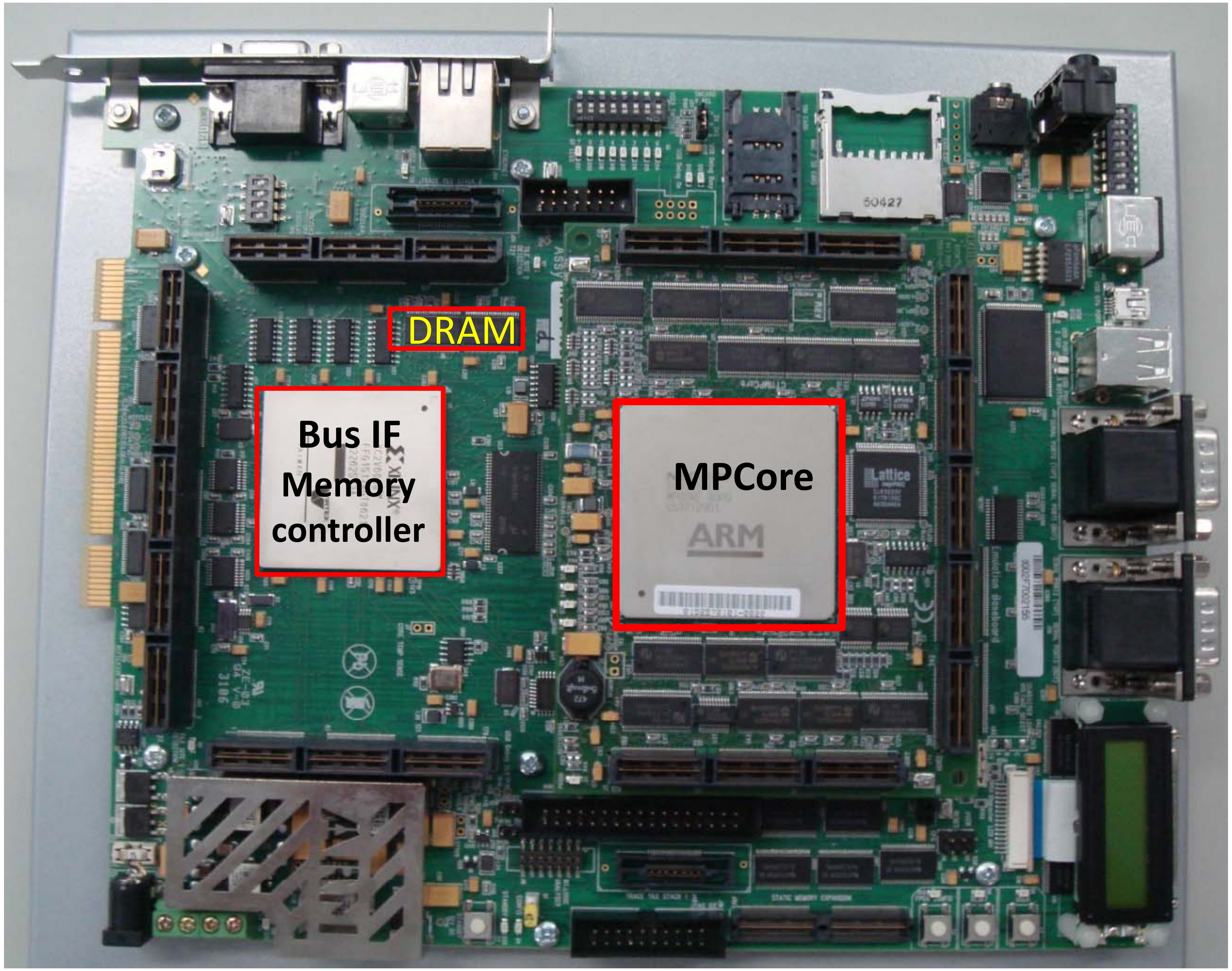
ARM+NEC

- ARM MP11 core (ARM11 architecture) x 4
  - ARMv6命令セット、ARM命令セット(32bit), Thumb命令セット(16bit), Jazelle命令セット(可変長)
  - 8-stage pipeline、1 instruction issue
  - L2 cache, 1MB,
    8way-set-assoc

- CT11MPCore +
  RealView Emulation
  Baseboardを使用
  - DDR-SDRAMコントローラ
    など周辺I/FはFPGAに搭載

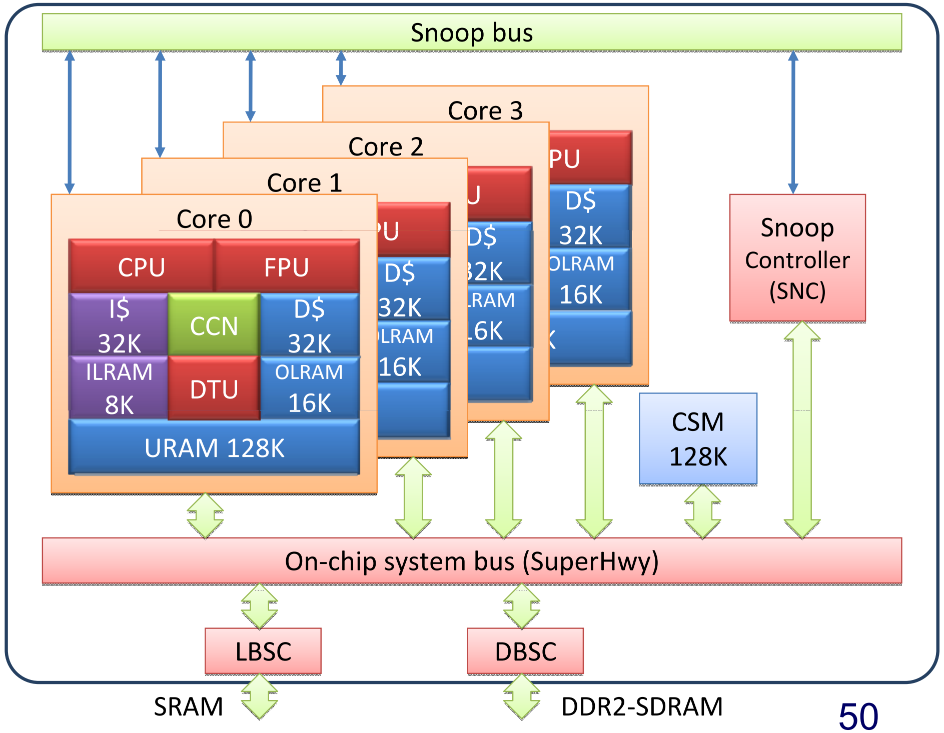


MPCore development kit



RP1 prototype

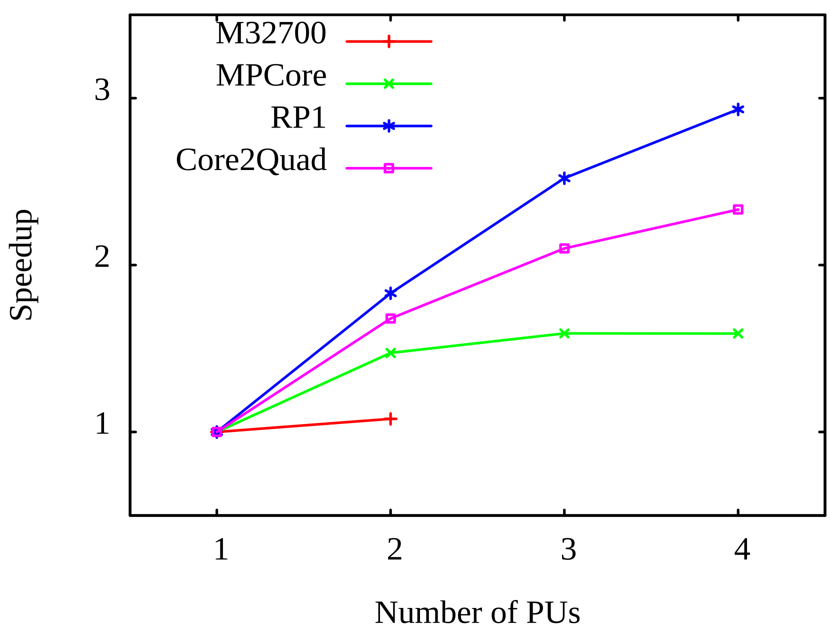
- SH-X3 architecture, SH-4A core x 4
  - 16bit命令セット，2命令同時発行可能
  - 8-stage pipeline
- Snoop Bus
  - SHwyのトラフィックを避けて転送
- On chip memory… (not used)
  - Local on-chip memory
    - 命令用 ILRAM (8Kbyte, 1clock)
    - データ用 OLRAM (8Kbyte, 1clock)
    - URAM (128Kbyte, 1～数クロック)
  - Shared memory(CSM, 128Kbyte)

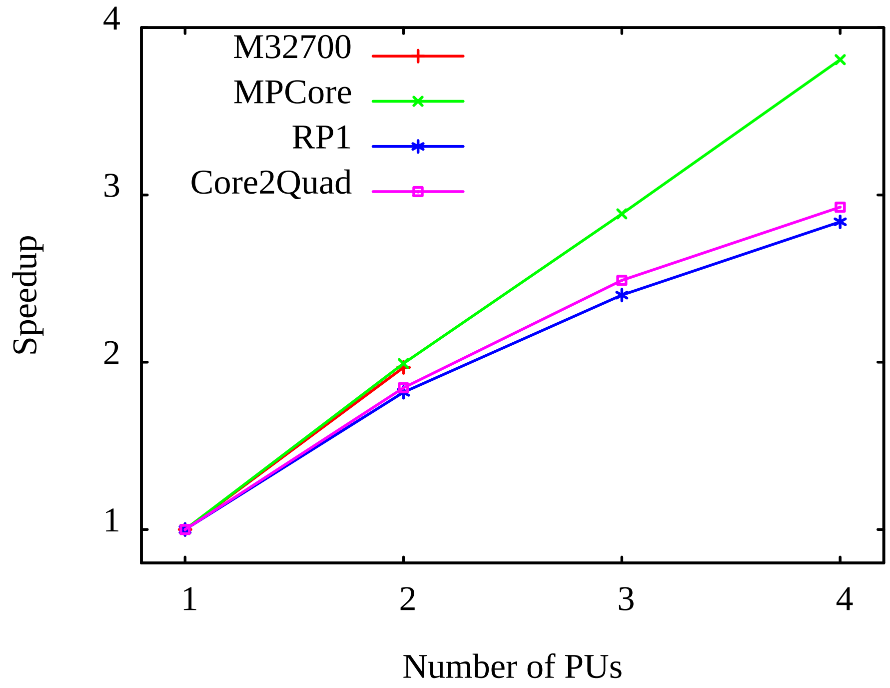


Comparison

	ルネサス M32700	ARM+NEC MPCore	早大+ルネサス+ 日立 RP1	Intel Core2Quad Q6600
#cores	2	4	4	4
Core frequency	300MHz	210MHz	600MHz	2.4GHz
Feq internal bus	75MHz	210MHz	300MHz	
Feq external bus	75MHz	30MHz	50MHz	
cache(I+D)	2way 8K+8K	4way 32K+32K L2, 1MB, 8way	4way 32K+32K	8way 32K+32K (L1) 16way 4M(2コア) x 2 (L2)
Line size	16byte	32byte	32byte	64byte
Main memory	32MB SDRAM 100MHz	256MB DDR-SDRAM 30MHz	128MB DDR2-600 300MHz	4GB DDR2-800 400MHz

NAS parallel benchmark: IS, CG

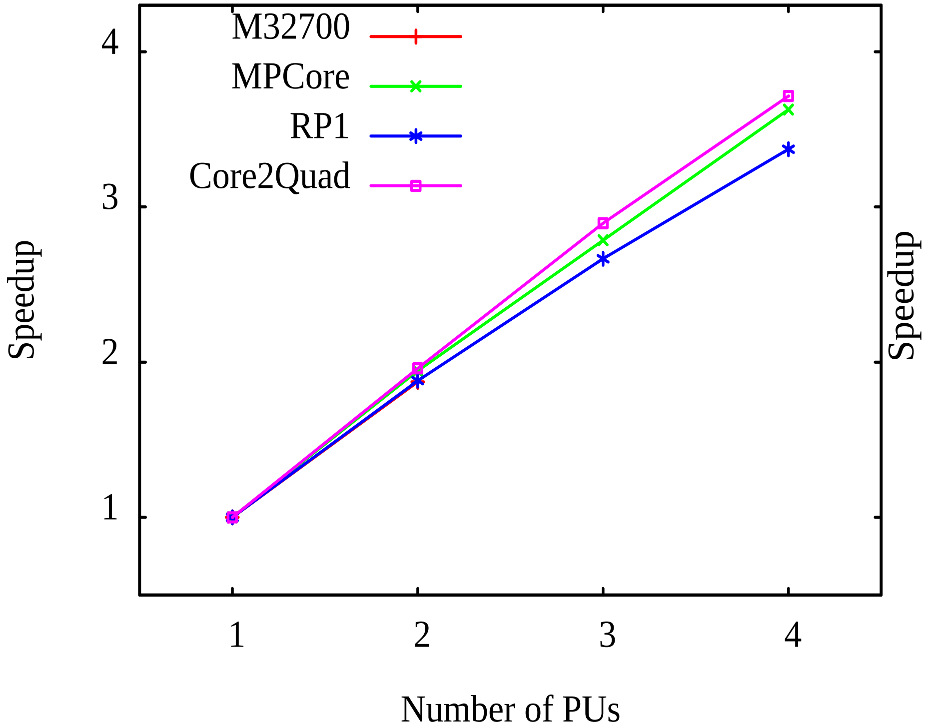




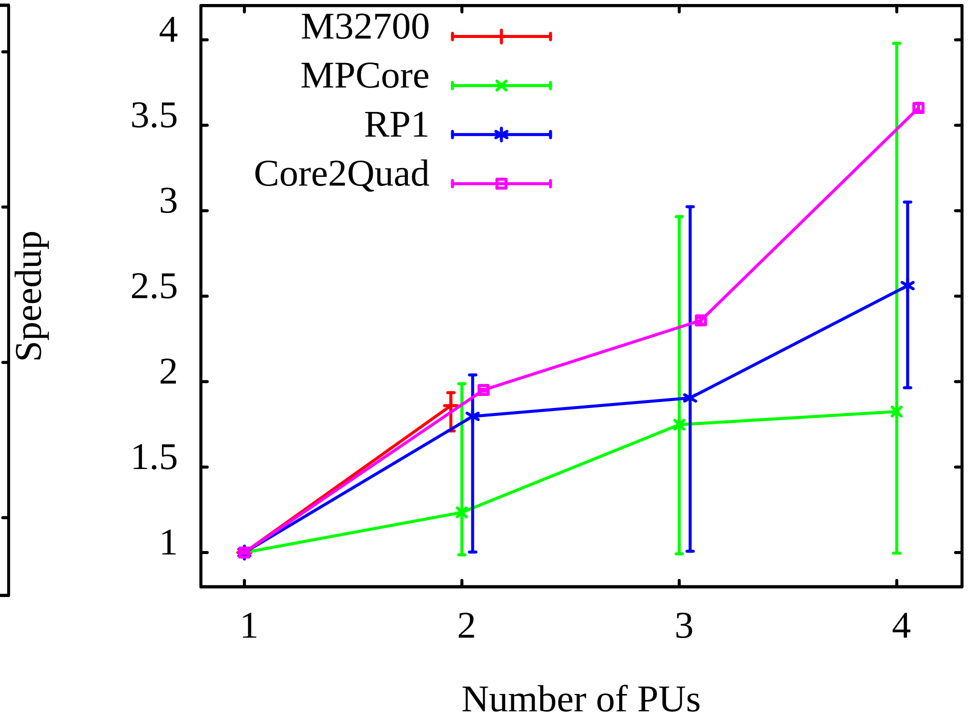
IS: Memory intensive、perf. Affected by memory bandwidth

CG: compute intensive

Susan smoothing 、BlowFish (ECBモード)



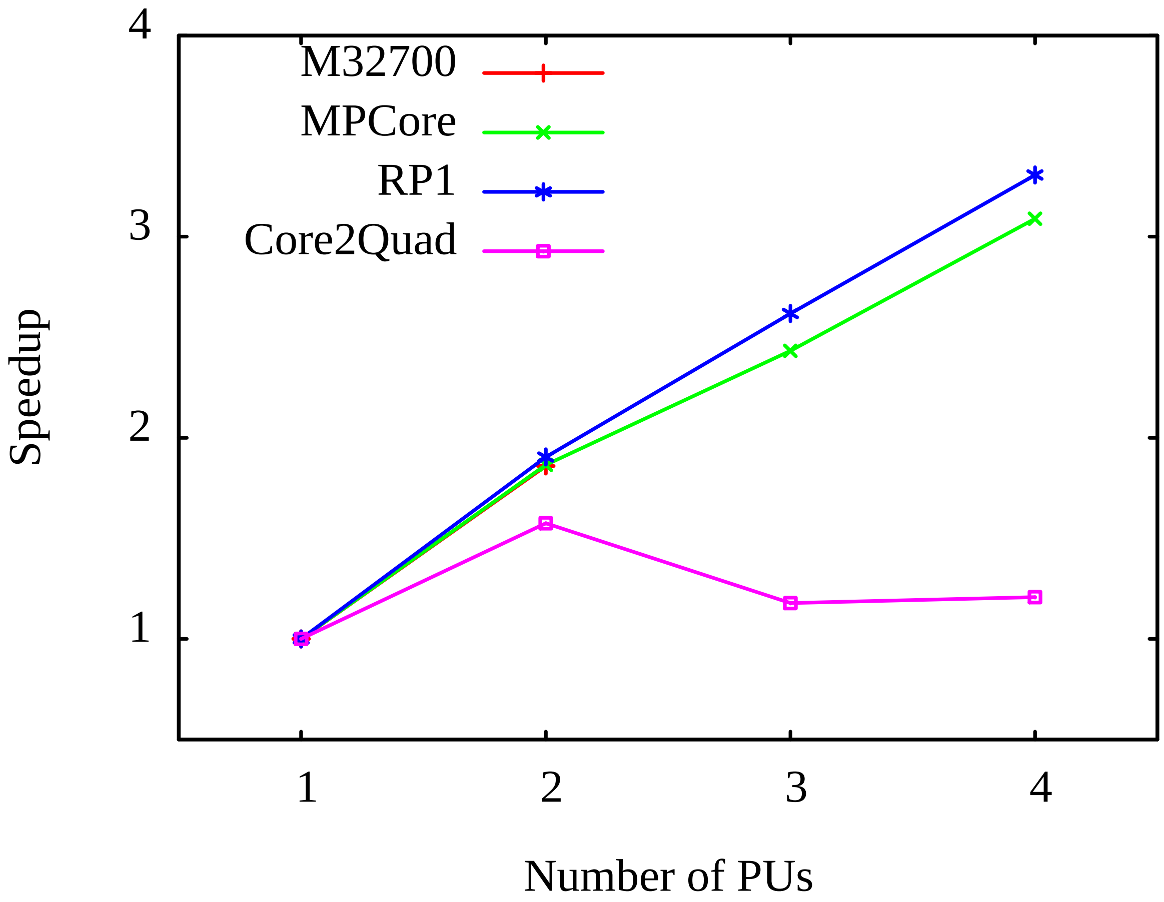
ばらつきが大きいため、error barで表
- error bar: 最大値と最小値、折れ線: 平均値



Scalable performance

File processing included
Core2Quad以外はNFS環境

FFT



Too short execution (a few mills sec) in case of Core2Quad => overhead too large

Programming Cost of parallelization by OpenMP

Parallelization by OpenMP

- Make parallel region large to reduce fork-join cost.
- Small modification from sequential

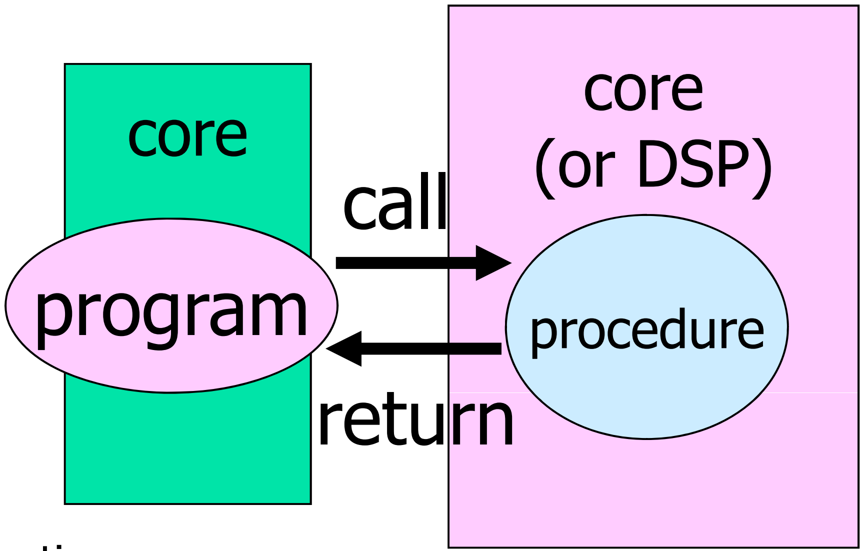
application	# of line added
susan smoothing	Directive 6 line added
Blowfish encoding	Directive 9 line added 12 line modified
FFT	Directive 4 line added
Mpeg2enc	Directive 5 line added 7 line modified

Programming for Multi-core processor by RPC

- RPC (remote procedure call)
  - Technology to execute some procedures in different memory space (usually, on remote computer)
  - Abstraction as a client-server(caller-callee), and hide complicated communication and protocol
    - Interface definition is described in IDL (interface description language), and stub for communication is generated automatically.
  - Technologies used in various applications.
    - SUN RPC – system programming
    - CORBA (common object broker arch)
    - GridRPC

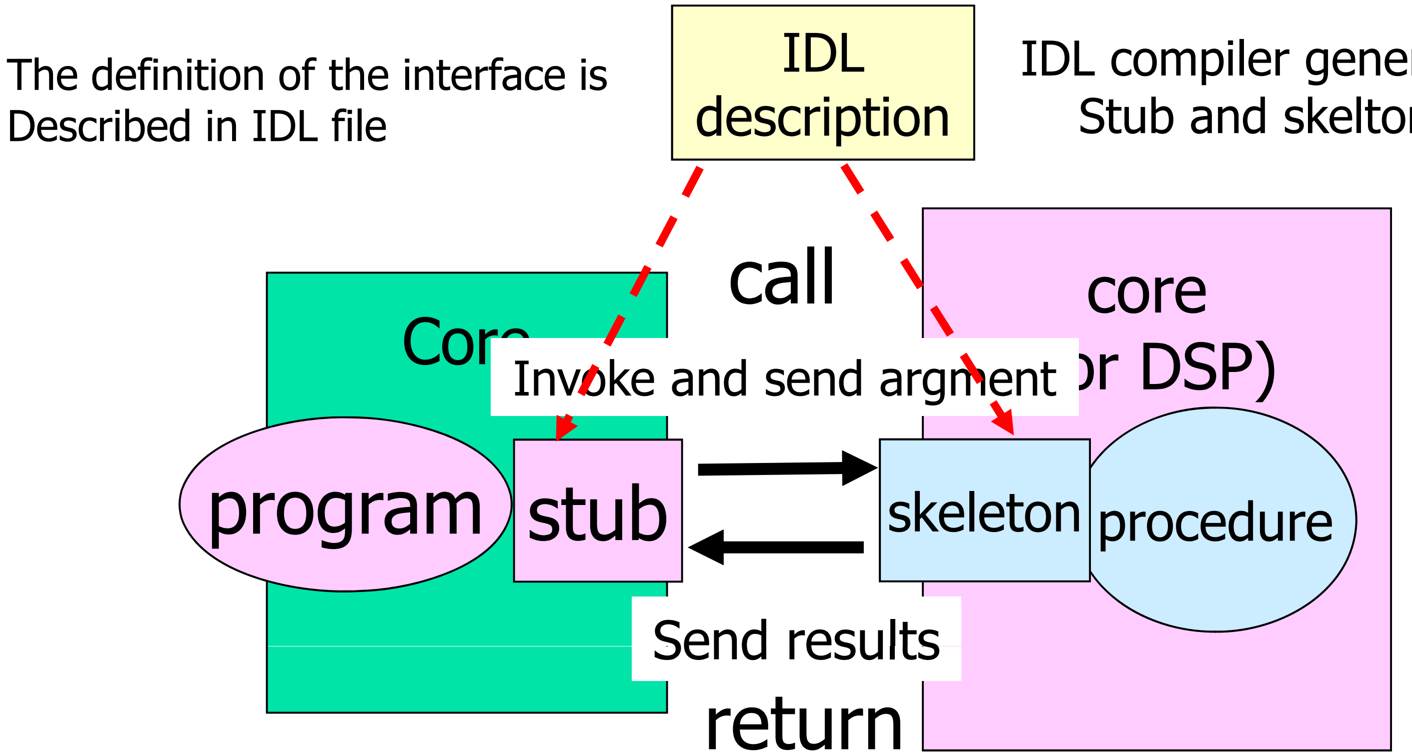
- RPC for multi-core processor
  - Assign functions to cores
  - Stright-forward abstraction for AMP
    - "Call some function as a RPC"
    - Also, it can be applied on SMP
  - It can be used for both shared memory

and distributed memory since it hide communication.



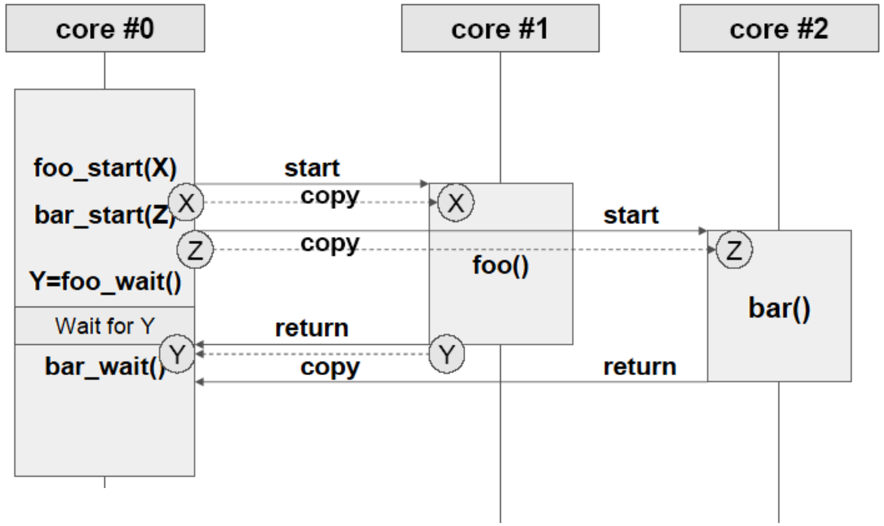
Mechanism of RPC

- Abstraction of client-server(caller-callee), hides detail protocol of communication
  - Interface is defined by IDL (interface description language), and generate communication by IDL compiler.
  - Stub – Called as a local function call and send argument/ recv results.
  - Skeleton – Accept call request and call the function in remote side.



Multi-core processor programming by Fujitsu Asynchronous RPC(ARPC)

- Fujitsu proposed Asynchronous RPC(ARPC) for Multi-core processor programming
- Asynchronous = multiple RPC requests can be executed in parallel

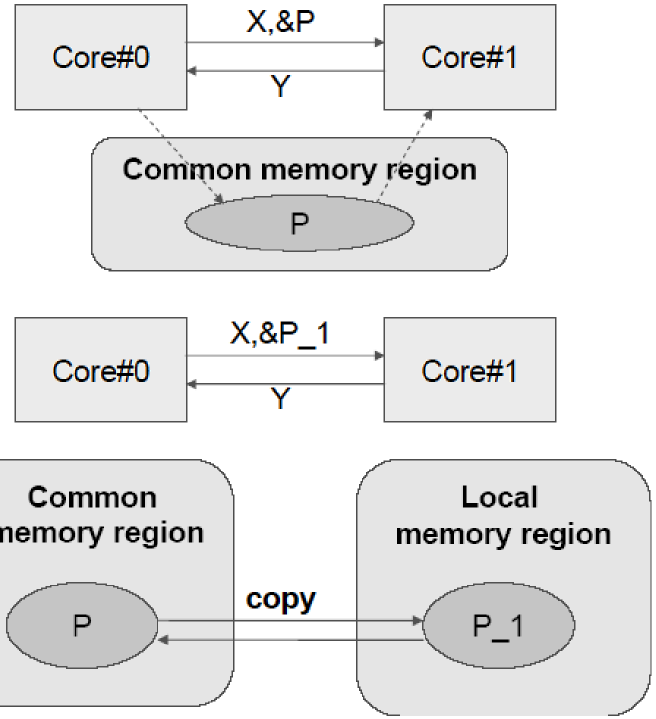


Common memory region

```
foo_start(&handle, X, &P);
…                 } Inhibit access to P
Y=foo_wait(&handle);
```

Local memory region

```
move_start(&handle1,
            &P_1, &P, size);
move_wait(&handle1);
foo_start(&handle2, X, &P_1);

…

Y=foo_wait(&handle2);
move_start(&handle1,
            &P, &P_1, size);
move_wait(&handle1);
```



- Easy to port code from sequential program.
- By hiding communication by RPC, portability is improved for several kinds of core incluing DSP.

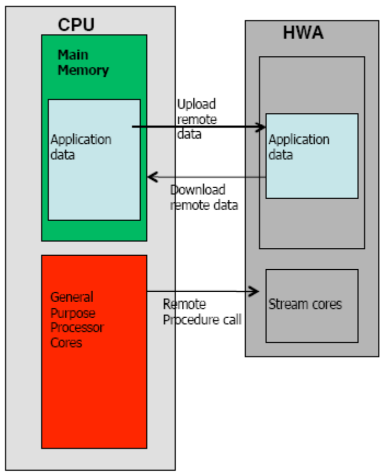
  ⇒ reduction of cost for development

Advanced multicore programming by RPC

- RPC is a good solution to use an original sequential program with small modification cost for several kind of processors.(AMP&SMP、DSP)
- Directive-base programming environment has been proposed
  - HMPP (hybrid multicore parallel programming)@INRIA
  - StarSs @BSC

```
#include <stdio.h>
#include <stdlib.h>

#pragma hmpp simple codelet, args[1].io=out
void simplefunc(int n, float v1[n], float v2[n], float v3[n], float alpha)
{
  int i;
  for (i = 0 ; i< n ; i++) {
    v1[i] = v2[i] * v3[i] + alpha;
  }
}
int main(int argc, char **argv) {
  unsigned int n = 400;
  float t1[400], t2[400], t3[400];
  float alpha = 1.56;
  unsigned int  j, seed = 2;
/* Initialization of input data*/
/* . . . */

#pragma hmpp simple callsite
  simplefunc(n,t1,t2,t3,alpha);

  printf("%f %f (...) %f %f \n", t1[0], t1[1], t1[n-2], t1[n-1]);
  return 0;
}
```

codelet / callsite directive set



Issues and agenda of programming environment for embedded multi-core processsors

- No standard, yet.
  - Embedded applications require several different kinds of configuration, so not easy to apply standard way to develop software.
  - Communication software for on-chip interconnect
    - MCAPI (Multicore Communication API)?
  - Standard (high-level) programming model and environment are prposed from high-end computing
    - ARPC ? OpenMP?
  - Multi-core processors for embedded will be distributed or shared memoy?
- Real time processing and parallel processing
  - Real-time scheduling with parallel task may be difficult (esp. in shared memory processor)
  - In real-time processing, parallel tasks needs multiple cores at a time.
  - Thread allocation fits to configuration of cores (core affinity)
    - sched_setaffinity is available form Linx 2.6, but it is mainly for HPC, not for embedded apps.
- Difficult debugging …