The Design, Analysis and Implementation of Algorithms for Parallel Shared Memory Machines

Casiano Rodríguez León

August 2, 2002

Preface

This draft is an attempt to collect, sort and write down the material generated teaching parallel programming for shared memory machines during the courses “Programación en Paralelo I” and “Computación” at Universidad de La Laguna.

Contents

1 An Introduction to OpenMP **5**
- 1.1 Installing OpenMP in your Computer 5
- 1.2 Hello cOMPunity! 5
- 1.3 Computing π 8
- 1.4 The *for* Work Sharing Construct 9
- 1.5 Dependencies 14
- 1.6 Matrix Multiplication 15
- 1.7 Generic High Performance Programming 15
- 1.8 Reductions 19

2 Shared Memory Machines **21**
- 2.1 Caches 21
- 2.2 SMP 23
- 2.3 Crossbars 24
- 2.4 ccNUMA 25
- 2.5 Routing 28
- 2.6 Cache Coherence: an Introduction 29
- 2.7 Directory Based Cache Coherency 30

3 Performance **33**
- 3.1 PAPI: A Portable Performance API 34
- 3.2 Measuring events in sequential programs with PAPI 35
- 3.3 Tuning 40
- 3.4 OpenMP and PAPI 43

A Programs **49**
- A.1 Generic Matrix Product and All Shortest Paths 49
- A.2 PAPI 52
  - A.2.1 Makefile 52
  - A.2.2 test_utils.c 53
  - A.2.3 do_loops.c 59
  - A.2.4 zero.c 61

Chapter 1

An Introduction to OpenMP

1.1 Installing OpenMP in your Computer

To get a shareware free distribution OpenMP compiler see [13] or [4]. Follow the instructions in the associated web pages.

1.2 Hello cOMPunity!

An OpenMP program begins execution as a single thread of execution called the *master thread*. The master thread executes in a *serial region* until the first *parallel construct* is encountered. In the OpenMP C/C++ Application Program Interface (API), the way to denote a parallel construct is through a *parallel directive*. A first approach to the syntax of a *parallel construct* is:

```
#pragma omp parallel [clause[[,]clause] ... ]
  structured-block
```

When a thread encounters a parallel construct, a *team* of threads is created. This thread becomes *master* thread of the team, with a thread number of 0, and all threads including the master thread, execute the region in parallel (see figure 1.1).

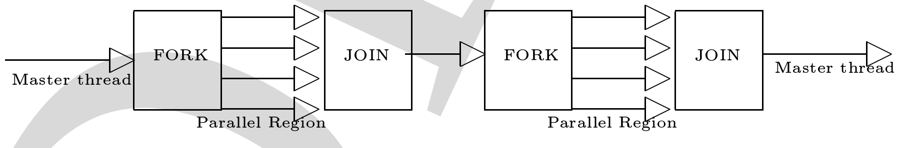

Figure 1.1: The OpenMP Fork/Join model

The environment variable `OMP_NUM_THREADS` defines the number of threads requested. This value can be changed calling the `omp_set_num_threads` function. The argument of the most recently executed call will be the number of threads requested.

The number of processors actually hosting the threads at any given time is implementation defined.

Once created, the number of threads in the team remains constant for the duration of the *parallel region*. The `omp_set_num_threads` function can be used to change the number of threads in a team from one parallel region to another.

By parallel region we understand the set of statements inside both the lexical and *dynamic extent* of the `structured-block`, that are executed by the multiple threads in the team. The concept of dynamic extent of a statement refers not only to all the statements that fall inside the lexical extent but also to any statement inside a function that is executed as a result of the execution of statements within the lexical extent. That is, all the statements that are "reachable" from `structured-block`. Actually, *dynamic extent* is a synonym of *parallel region*.

The concept of *structured-block* refers to a single or compound statement that has a single entry point and a single exit point. No statement is a structured block if there is a jump into or out os that statement (that includes a `longjump()` or the use of `throw`, but a call to `exit` is permitted). A compound statement is a structured block if its execution always begins at the opening `{` and always ends at the closing `}`.

The statements contained within the dynamic extent of the parallel region are executed by each thread.

There is an implied synchronization *barrier* at the end of the parallel region. A barrier is a point that must be reached by all the threads in the team. Each thread waits until all the other threads arrive at this point. The master thread of the team continues execution at the end of the parallel region.

The "Hello World" listing 1.1 on the next page illustrates some of these concepts.

The function `omp_get_thread_num()` returns the number associated with the thread. Each thread has a different number between 0 and the number of threads minus one. Another function, `omp_get_num_threads()` returns the number of threads in the current team.

Let us see the compiling and execution of the listing 1.1 on the facing page. The environment variables related with OpenMP in my computer are defined as follows:

```
> env | grep OMP
OMP_NUM_THREADS=2
OMPC_NUM_PROCS=16
```

The following two lines show the compilation (with the omni compiler) of the program 1.1 on the next page, asuming it is saved in a file called `phello.c`:

```
> ompcc phello.c -o phello
Compiling 'phello.c'...
```

See the output in execution 1.1 on the facing page.

Listing 1.1 OpenMP Hello World program

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
main() {

  printf("Hello world. Number of threads determined by \
                       the environment variable ...\n");
#pragma omp parallel
  {
    printf("Hello world from %d\n",omp_get_thread_num());
  }
  printf("Hello world. Let us work with 4 threads...\n");
  omp_set_num_threads(4);
#pragma omp parallel
  {
    printf("Hello world from %d\n",omp_get_thread_num());
  }

  printf("Back to the sequential world\n");

  printf("Now 3 threads ...\n");
  omp_set_num_threads(3);
#pragma omp parallel
  {
    printf("Hello world from %d\n",omp_get_thread_num());
  }

  printf("Back to the sequential world\n");
}
```

```
> phello
Hello world. Number of threads determined by the environment
variable ...
Hello world from 1
Hello world from 0
Hello world. Let us work with 4 threads...
Hello world from 2
Hello world from 1
Hello world from 3
Hello world from 0
Back to the sequential world
Now 3 threads ...
Hello world from 1
Hello world from 2
Hello world from 0
Back to the sequential world
```

Execution 1.1: Execution of the hello world program in fig. 1.1

1.3 Computing π

The area under the curve $y = \frac{1}{1+x^2}$ between 0 and 1 provides a way to compute π:

$$\int_0^4 \frac{4}{(1+x^2)}dx = 4\arctan(x)|_0^1 \ = 4(\frac{\pi}{4} - 0) = \pi$$

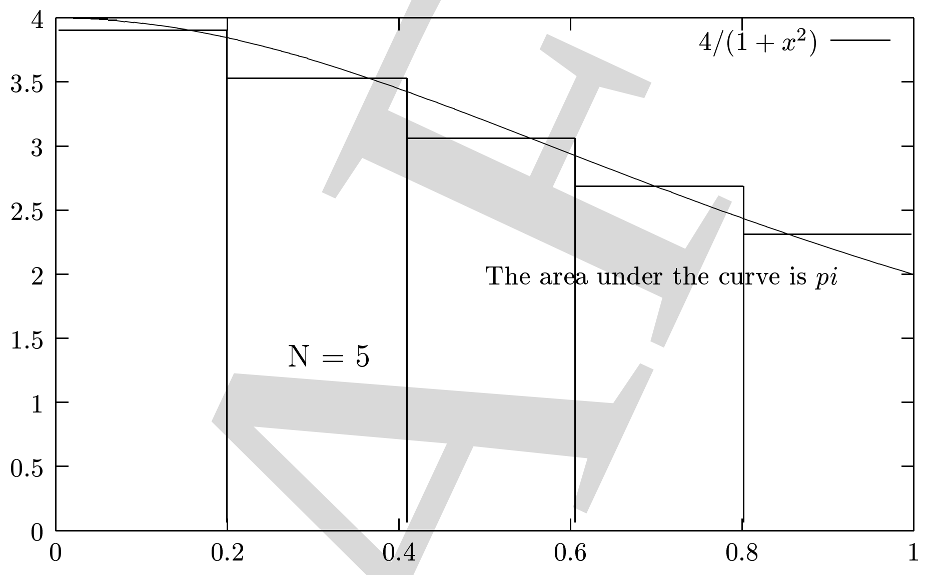

Figure 1.2: $\pi = \int_0^4 \frac{4}{(1+x^2)}dx$

This integral can be approached by the sum:

$$\pi \simeq \sum_{i=0}^{N-1} \frac{4}{N \times \left(1 + (\frac{i+0.5}{N})^2\right)} \tag{1.1}$$

From this formula follows a straighforward strategy to have a parallel program computing π.

1. Let be id the identifier of each "thread" and P the number of threads.

2. Each thread id computes in a private variable π_{id} the sum of roughly N/P summands.

3. Then thread 0 computes $\simeq \pi_0 + \cdots \pi_{P-1}$ and saves its value in the shared variable π.

OpenMP provides all the constructs to express such algorithm in a simple and comfortable way. The feature we need to write this program is the capability to differentiate between shared and private variables. When a variable x is scoped "private" that means that there is a memory storage space for x per thread, and each thread, when refering to x is refering to its associated storage space for x. On the opposite side, a shared variable y names a single block of storage. When refering to y, all the threads will access the same block of storage.

Among the clauses that can be used with the parallel construct, one of the most important is the *private* clause.

```
#pragma omp parallel private( variable-list ) ...
  structured-block
```

Most of the clauses accept a *variable-list* argument, which is a comma-separated list of variables that are visible. The `private(i, local, id, pi_id)` declaration at line 22 in program 1.2 on the following page is an example. All the variables that appear within directive clauses must be visible. Of course, there is also a a *shared* clause with similar syntax. There is also a *default* clause that allows the user to affect the data sharing attributes of variables. The syntax is:

```
default ( shared | none )
```

Specifying `default (shared)` is equivalent to explicitly listing each currently visible variable in a `shared` clause. In the absense of an explicit `default` clause, the default behavior is the same as if `default (shared)` were specified. Therefore, the remaining variables `pi, w, pi_, N` and `num_threads` in program 1.2 on the next page are shared. Specifying `default(none)` requires an explicit declaration of the data sharing attribute of each variable.

In lines 21 to 29 each thread accumulates in the private variable `pi_id` a diferent subset of indexes of the sum in equation 1.1 on the facing page. To "communicate" its partial result to the other threads, that value is copied into the shared variable `pi_[id]` (line 30).

Since the end of the parallel construct carries an implicit synchronization barrier, we know that when thread 0 reaches line 33, all the threads in the former parallel construct have completed the assignment in line 30. Thread 0 performs the final task of adding the (relatively few) `num_threads` summands to obtain the approach to π.

1.4 The *for* Work Sharing Construct

As we have seen in section 1.3 on the preceding page the threads in the team created by a parallel construct execute the statements in the dynamic extent of the associated parallel region. There are, however, a few OpenMP constructs that involve the implicit branching of the threads in the team. The general idea behind being that there is some work to do, and each thread is going to do a different chunk of work. This is why these constructs are collectively known as *Work Sharing Constructs*. An example is the *for* directive. Through the *for* directive, the OpenMP programmer indicates that the iterations of the following *for* loop have to be distributed (partitioned) across the threads in the current team. The syntax of the *for* loop is as follows:

```
#pragma omp for [clause[[,] clause] ...  ]
  for(init-expr; var logical-op b; incr-expr)
    structured-block
```

Listing 1.2 Using shared and private variables

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

main(int argc, char **argv) {
  double local, pi = 0.0, pi_id, w;
  double *pi_;
  long N, i, id, num_threads;

  if (argc > 2) {
    num_threads = atoi(argv[1]);
    pi_ = (double *) malloc(num_threads*sizeof(double));
    omp_set_num_threads(num_threads);
    N = atoi(argv[2]);
  }
  else {
    fprintf(stderr,"Usage is:\n%s numthreads numintervals",argv[0]);
    exit(1);
  }

  w = 1.0 / N;

#pragma omp parallel private(i, local, id, pi_id)
  {
    pi_id = 0.0;
    id = omp_get_thread_num();
    for (i = id; i < N; i += num_threads) {
      local = (i + 0.5)*w;
      pi_id = pi_id + 4.0/(1.0 + local*local);
    }
    pi_[id] = pi_id;
  }

  for  (i = 0; i < num_threads; i++)
    pi += pi_[i];

  printf("pi is %1.20f\n", pi*w);
}
```

Any work sharing construct must be encountered by all threads in a team, and the statements within the structured block are executed by one or more threads. there is an implicit synchronization barrier at the end of the work sharing construct.

Not any *for* arbitrary loop is allowed. The *init-expr* has to be an assignment to the *var* mentioned on the left of the *logical-op*:

$$var = lb$$

The *var* has to be an integer variable (in the next version of OpenMP *var* will be implicitly made private for the duration of the *for*, but be aware of the behavior of current OpenMP compilers). The *logical-op* has to be one of `<`, `<=`, `>`, `>=` and the *incr-expr* has to be one of the following:

- ++var
- - -var
- var- -
- var += incr
- var -= incr
- var = var + incr
- var = incr + var
- var = var - incr

Fundamentally, *lb*, *b* and *incr* have to be *loop invariant integer expression.* That is, their values have to be known at the entry of the loop. There is no synchronization during the evaluation of these three expressions. Thus, any evaluated side effects produce indeterminate results. The values of the loop control expressions must be the same for all the threads in the team. A loop written following these conventions is said to be in canonical shape.

Observe the program 1.3 on the following page and the corresponding execution 1.2 on page 13.

The first loop directive at line 8 produces 4 "hellos", since each of the 2 existing threads execute 2 iterations. Line 17 shows the syntax of the shortcut provided by OpenMP for the common situation when a *parallel* directive is followed by a *for* directive. The syntax of the *parallel for* directive is as follows:

```
#pragma omp parallel for [clause[[,] clause] ...  ]
  for(init-expr; var logical-op b; incr-expr)
    structured-block
```

The *for* loop at line 27 is completely replicated by the two threads producing 8 "hellos".

Listing 1.3 There will be $4 + 4 + 4 \times \mathit{omp\_get\_thread\_num}()$ “hellos”

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
main() {
  int i;

  printf("-- parallel followed by for\n");
#pragma omp parallel private(i)
#pragma omp for
  for(i=0; i< 4; i++)
  {
    printf("Hello world from %d i=%d\n",omp_get_thread_num(),i);
  }

  printf("-- Abbreviated syntax\n");

#pragma omp parallel for private(i)
  for(i=0; i< 4; i++)
  {
    printf("Hello world from %d i=%d\n",omp_get_thread_num(),i);
  }

  printf("-- parallel only\n");

#pragma omp parallel private(i)
  {
    for(i=0; i< 4; i++)
    {
      printf("Hello world from %d i=%d\n",omp_get_thread_num(),i);
    }
  }

}
```

```
> echo $OMP_NUM_THREADS
2
> ompcc wshello.c
Compiling 'wshello.c'...
> a.out
-- parallel followed by for
Hello world from 0 i=0
Hello world from 1 i=2
Hello world from 0 i=1
Hello world from 1 i=3
-- Abbreviated syntax
Hello world from 0 i=0
Hello world from 1 i=2
Hello world from 0 i=1
Hello world from 1 i=3
-- parallel only
Hello world from 0 i=0
Hello world from 1 i=0
Hello world from 0 i=1
Hello world from 1 i=1
Hello world from 0 i=2
Hello world from 1 i=2
Hello world from 0 i=3
Hello world from 1 i=3
>
```

Execution 1.2: Execution of the program in fig. 1.3 on the facing page

1.5 Dependencies

The *for* directive simplifies the parallelization of C loops where the different iterations are "*independent*". That means that there are no "*race conditions*" among the different iterations of the loop. There is a dependence if, for example, an iteration of the *for* loop, say i_1, writes in variable a_k and another iteration i_2 reads the value of such variable a_k. To preserve the semantic of the sequential loop, iteration i_2 has to wait for iteration i_1 to finish its writing of variable a_k.

But the OpenMP execution goes asynchonously, and therefore there are no guarantees about the execution order of the iterations inside the loop. Thus, the value read by i_2 depends on the speed of the two involved threads. If the the thread executing iteration i_2 arrives first, the read value will be the wrong one. That is why it is called a "race condition": depending on the speed of the threads you can get different results.

Since dependences are always associated with a particular memory location, they can be detected analyzing the use of each variable inside the loop:

- Is the variable only read and never assigned within the loop body? if so, there are no dependences involving it.
- Otherwise, consider the memory locations that make up the variable and that there are assigned within the loop. for each such location, if there is exactly one iteration that accesses that location we can conclude that there are no dependences related with that variable.

More formally, assume you have a loop like:

```
     for(i = 0; i<n; i++) {
T1:      ...
T2:      ...
         ...
Tk:      ...
     }
```

where, for sake of simplicity, all the statements T^j are assignment statements. Let be T_i^j the execution in iteration i of statement T^j and let us denote by $\overline{v}$ the set of memory locations allocated for variable v. Let be $M(T_i^j)$ the set of memory locations accesed during T_i^j. There is a dependence if:

$$\exists i_1 \neq i_2 \in \{0, n-1\} \quad \exists j_1, j_2 \in \{1, k\} : M(T_{i_1}^{j_1}) \cap M(T_{i_2}^{j_2}) \neq \emptyset \tag{1.2}$$

1.5.1 Example

As an example, consider the following loop:

```
     for(i = 1; i<n; i+=2) {
T1:    a[i] = a[i] + a[i-1];
     }
```

It is: $M(T_i^1) = \{\overline{a_i}, \overline{a_{i-1}}\}$. Therefore the equation 1.2 becomes:

$$i_1 \neq i_2 \wedge M(T_{i_1}^1) \cap M(T_{i_2}^1) \neq \emptyset \Rightarrow i_1 \neq i_2 \wedge \{\overline{a_{i_1}}, \overline{a_{i_1-1}}\} \cap \{\overline{a_{i_2}}, \overline{a_{i_2-1}}\} \neq \emptyset \tag{1.3}$$

these two equations have as only solution $i_1 = i_2 - 1$ and so $i_2 = i_1 + 1$ with $i_1, i_2 \in I = \{1, 3, 5, \dots\}$. This equation has no solution in the set of indices I and so we can conclude that the former loop is independent and a *for* directive can be attached to it.

1.6 Matrix Multiplication

Loop nests can contain more than one loop, and arrays can have more than one dimension. The three-deep loop nest in listing 1.4 on the following page computes the product of two matrices $C = A \times B$, following the classical formula:

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{k,j} \tag{1.4}$$

In this case the index space is the cube

$$[0, N-1]^3 = [0, N-1]_i \times [0, N-1]_j \times [0, N-1]_k \tag{1.5}$$

However, we are now interested in the parallelization of the outer loop at line 40. There are several assignments inside this loop: to `i` (line 40 `i = 0` and `i++`), `j` (line 41), `sum` (line 42) and `k` (line 43). Since we are declaring all these variables `private`, there is no conflict related with them and each thread uses its own copy.

The acceses to `a[i,k], b[k,j]` at line 44 are always for reading and therefore they do not introduce any dependence conflict.

Last, during the assignment statement T^{45} to `c[i][j]` at line 45, threads in charge of different iterations i write in different locations

$$M(T_i^{45}) = \{\mathtt{c_{i,j}} | \mathtt{j} \in \{\mathtt{0} \dots \mathtt{N} - \mathtt{1}\}\}$$

and so $M(T_{i_1}^{45}) \cap M(T_{i_2}^{45}) = \emptyset$ if $i_1 \neq i_2$.

1.7 Generic High Performance Programming

The title of this section combines two conflicting concepts: Generic Programming and High Performance Computing (HPC). If you want to make it generic then you'll have to pay with some (usually not negligible) loss of performance and ... viceversa. *Object Oriented Programming* (*OOP*) is the most used approach to generic programming. However, we have seen that often HPC programmers are reluctant to accept the trade, since they feel the loss in performance introduced by OOP is excessive to justify the gain in reusability. An alternative is an structured use of macros as we'll show in the next paragraph.

1.7.1 Example

The matrix product function in program 1.4 on the next page was written to work with any arbitrary commutative semi-ring $(\oplus, \odot)$ instead to be restricted to the classic $(+, \times)$ operations. A useful example is the semi-ring $(min, +)$. To

Listing 1.4 Matrix product

```
void m_mat(matrix c, matrix a, matrix b) {
  int i, j, k;
  double sum;

#pragma omp parallel
#pragma omp for private(i, j, k, sum)
  for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
      sum = m_neutral;
      for(k = 0; k < N; k++)
        sum = m_plus(sum, m_times(a[i][k], b[k][j]));
      c[i][j] = sum;
    }
  }
}
```

emphasize the semi-ring structure, we will use the "operational" notation for *min* instead its classic functional notation:

$$a\ min\ b = \begin{cases} a & \text{if } a \le b \\ b & \text{otherwise} \end{cases} \tag{1.6}$$

The distributive property applies:

$$a + (b\ min\ c) = (a + b)\ min\ (a + c)$$

and the neutral element for *min* is ∞:

$$a\ min\ \infty = \infty\ min\ a = a$$

Program 1.5 on the facing page show the set of macros defining the semiring $(min, +)$ so that the function in listing 1.4 produce as result:

$$c_{i,j} = \min_{k=0}^{N-1}(a_{i,k} + b_{k,j}) \tag{1.7}$$

One of the applications of this definition is the following: Given a graph G with n vertices through its $n \times n$ adjacency matrix $A = (a_{i,j}) = (distance(i,j))$, the *minimum distance* (*shortest path*) between to vertices i and j is given by the $(min, +)$-power matrix A^n. The listing in 1.6 on page 18 shows the corresponding code.

See appendix A.1 on page 49 for the complete program.

Observe that, since the *Dynamic Programming* technique consists in transforming a problem in a shortest path problem, the program in listings 1.5 on the facing page, 1.4 and 1.6 on page 18 can be considered a primitive skeleton of a dynamic programming solver.

Listing 1.5 Declarations for the matrix product

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

/* N has to be a power of two */
#define N 4
#define MAX_DIMENSION 32

#define classic

#ifdef classic
/***** the add mul object *********/
...
/****** end add mul object ********/
#else
/***** the min add ********************/
#define m_type unsigned
#define m_neutral ULONG_MAX
#define m_plus(a,b) (((a)<(b))?(a):(b))
#define m_times(a,b) ((a)+(b))
#define m_format_in "%d"
#define m_format_out "%d\t"
/***** end min add object ***********/
#endif

typedef m_type matrix[N][N];

matrix a;
matrix temp;
matrix c;
```

Listing 1.6 The matrix power function

```
void m_power(matrix c, matrix a, int k) {

  m_assign(temp, a);
  m_mat(c, temp, temp);
  k /=2;
  while(k > 1) {
    m_assign(temp, c);
    m_mat(c, temp, temp);
    k /=2;
  }
}

int main() {
  int i, j, k;


    m_init();
    printf("A\n");
    printMat(a);

    m_power(c, a, N);

    printf("C\n");
    printMat(c);
}
```

1.8 Reductions

From the perspective of a parallel programmer, the former problem of computing π belongs to a remarkable general class of problems called *reduction problems.* In a reduction problem, you have a vector of values x_i with $i = 0, \cdots N-1$, and a associative and commutative operation $\circ$ and you have to compute the value:

$$\bigcirc_{i=0}^{N-1} x_i = x_0 \circ \cdots \circ x_{N-1} \tag{1.8}$$

Obviously, the technique applied in program 1.2 on page 10 can be applied to compute in parallel $\bigcirc_{i=0}^{N-1} x_i$. As a shortcut to the program in 1.2 on page 10, OpenMP provides the concept of `reduction` variables. Observe program 1.7 on the next page and compare it with program 1.2 on page 10. The use of the clause `reduction` greatly simplifies the algorithm.

The syntax of the `reduction` clause is:

`reduction (`*`op : variable-list)`*

A private copy of each variable x in *variable-list* is created, one for each thread x_t, as if the `private` clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the operator specified:

$$x = x_{old} \circ \bigcirc_{t=0}^{P-1} x_t \tag{1.9}$$

The operator is used to determine the initial value for the private copies, that is, to the identity e of such operator $\circ$,

$$x_t = e$$

One caveat about paralelizing reductions is that when the type of the reduction variable is floating point, the final result may not be precisely the same as when performed serially. The reason is that, although the sum of real number is commutative and associative, the sum of float numbers with finite precision is not. For example, we add up four floating point numbers that are accurate to four decimal digits. If the numbers are added up in this order:

$$\begin{aligned}((0.0004 + 1.000) + 0.0004) + 0.0002 &= (1 + 0.0004) + 0.0002\\ &= 1 + 0.0002 = 1\end{aligned}$$

When added in ascending order we get a more accurate result:

$$\begin{aligned}((0.0002 + 0.0004) + 0.0004) + 1.000 &= (0.0006 + 0.0004) + 1.000\\ &= 0.001 + 1 = 1.001\end{aligned}$$

The user have to be aware of the differences between the serial and parallel versions due to roundoff.

Listing 1.7 Using the reduction clause to compute π

```
#include <stdio.h>
#include <omp.h>
#define N 100000000

main() {
  double local, pi=0.0, w;
  long i;

  w = 1.0 / N;

#pragma omp parallel private(i, local)
  {
#pragma omp single
    pi = 0.0;
#pragma omp for reduction (+: pi)
    for (i = 0; i < N; i++) {
      local = (i + 0.5)*w;
      pi = pi + 4.0/(1.0 + local*local);
    }
  }

  printf("pi is %1.20f\n", pi*w);
}
```

Chapter 2

Shared Memory Machines

2.1 Caches

A cache is nothing but fast and efficient memory. However cache-memory is also more expensive and therefore only small amounts of it are used when designing and building a system.

While using your computer, all active programs are stored in active memory (RAM). The parts of the memory used very recently are stored in the cache memory. When data is needed for the CPU it will first look in the fast cache memory, before looking in the slower main memory.

If the CPU needs to do some calculations with the same group of data, it's likely that the required information will be stored in the cache. In such cases the CPU will not need to access the slower RAM.

There are different kinds of cache. Every CPU has a small amount of cache in the CPU. It's called Level 1 (L1) cache because it's the first place the CPU will look for information. Besides the L1 cache there's often a second layer of cache, called level 2 cache (L2). This cache is slower than the L1 cache but still much faster than regular memory.

If information isn't found in the L1 cache the CPU will then look in the L2 cache. If the information is still not found the CPU will finally get the information from the main memory. The main memory is divided into groups of consecutive words called cachelines. In figure 2.1 on the next page each cacheline is represented by a stack of squares. Memory cells are moved into the cache in cacheline units.

If a value is needed twice, or if another value in the same cacheline is needed, the the CPU can take the needed value directly from the cache. The first situation is referred as *temporal locality* while the second is considered *spacial locality*. A program showing spacial and temporal localities will often find its date in the cache (a *cache hit*). The opposite of a cache hit is a *cache miss*: a new cacheline has to be loaded from main memory. If a value in a cacheline is changed the cacheline it must eventually be written back to memory. The two most common policies are *write through* and *write-back*.

- In a write through cache, the data is inmediately written back to memory. Memory and cache are always consistent.

- In a write-back cache a bit is added to each entry in the cache indicating when the memory is "dirty". When the system "evicts" an entry from the cache, the systems checks the bit. If it is dirty, main memory is updated.

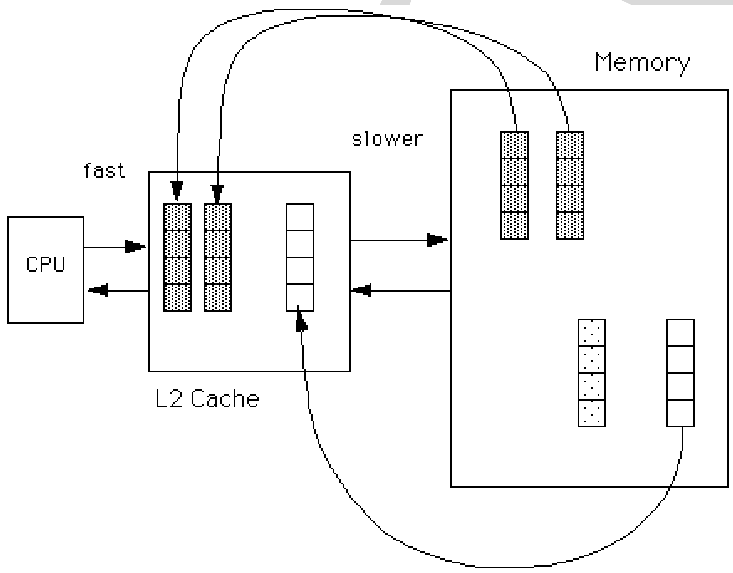

Figure 2.1: Scheme of a cache

2.1.1 Example

To understand the concepts, let us have a more detailed look into a very simplified cache. Assume that we have a 32-bit addressed memory and that each address refers to a byte in memory. Since $2^{32} = 2^2 \times 2^{10} \times 2^{20} = 4GB$, we have $(4 \times 1024)MB = 4GB$ addresses.

Assume a cache organized as $8192 = 2^{13}$ entries (figure 2.2 on the facing page).

An entry in the cache is as follows:

- One bit to say whether the entry is *dirty cache entry* or *clean cache entry* (i.e. if the content of the cache is coherent with the corresponding cell in main memory).
- Many addresses map into the same location in the cache. In fact, all the 2^{16} addresses with the same index map in the same cache entry. Therefore, there are 65536 addresses sharing the same entry. The 16 bit *cache tag field* contains the upper 16 bits of the address present in the cache entry. With this information the system will be able to reocgnize which address we are talking about.
- The 8 bytes of containing the data

Each entry takes 8 bytes for the data plus 2 bytes for the tag and one bit for the *dirty* bit. The size of the table is $8192\ entries \times ((8+2) \times 8 + 1)\ bits = 8192\ entries \times 81\ bits = 663552\ bits$.

A memory address is divided in:

Tag (bits 31 to 16)	It is saved into the tag field in the cache
Index (bits 15 to 3)	To index into the table
Offset (bits 2,1 and 0)	To access the byte within the 8 bytes

Every time the processor makes a memory reference, the algorithm used is as follows:

- The system takes bits 3 to 15 to index into the cache. Given the cache entry, it compares the tag with the upper 16 bits of the address. If they match, there is a *cache hit*, if not there is a *cache miss*.
- In the case of a cache miss, the system goes to the memory to find the data, that data replaces the one currently in the cache. If the current data is dirty, it must first written out to memory. If not, the system throws it out.

To provide spacial locality, entries in a cache are tipically from 32 to 256 bytes. The set of entries in a single cache location is what we have called a *cache line*.

In this example, each address maps into only one location in the cache. This is called a *direct mapped cache*. The opposite is a *set associative cache*. In a set associative cache a memory address can map into more than one table entry. Most nowadays caches are associative.

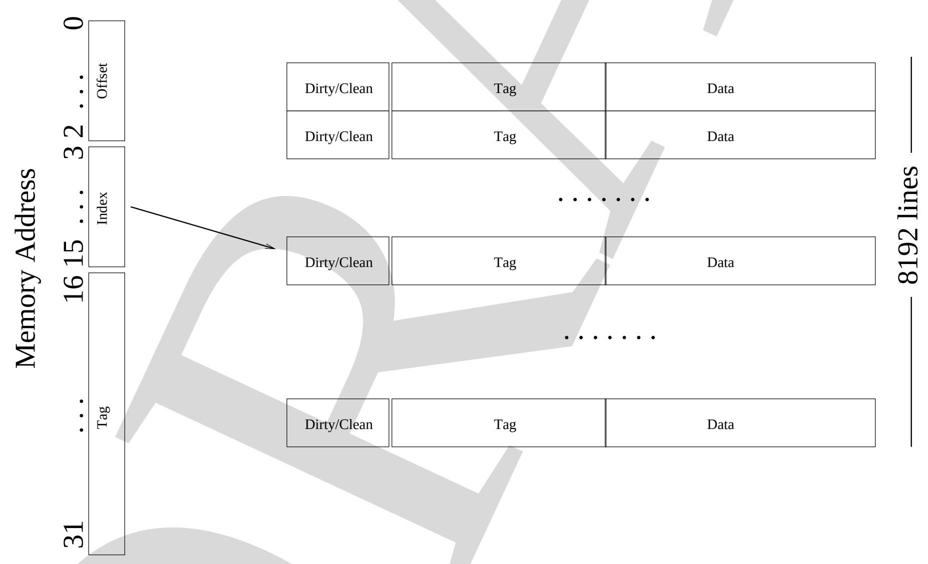

Figure 2.2: Simplified Cache

2.2 SMP

The concept of *symmetric multiprocessing* (*SMP*) has trickled down from expensive supercomputer hardware all the way to today's desktop workstation.

More and more personal desktop systems are appearing with capacity for up to four processors, and modest server systems regularly contain four, eight, sixteen or more CPUs. The beauty of the SMP model is that all processors share the same memory areas, so to the user the system appears just like a “regular” desktop system.

SMP “nodes” can also be used as building blocks for larger *High Performance Computing HPC* systems; instead of networking N processors together, vendors can network M nodes, where each node contains p processors sharing the node memory over a high-performance bus or crossbar.

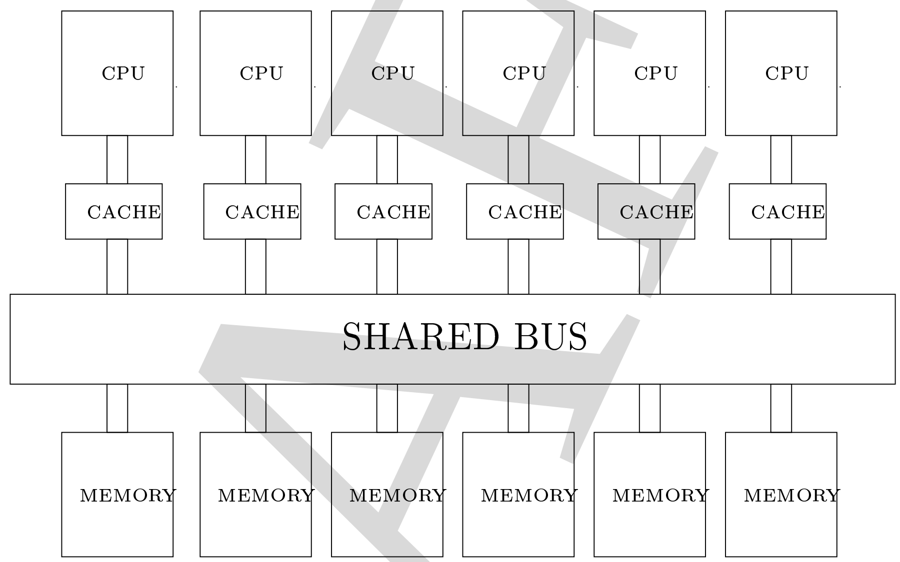

Figure 2.3: bus-based SMP architecture

In standard SMP architectures there are, typically two methods used in the coupling of processors and memory, a bus-based architecture or a crossbar. A bus-based system looks very much like the standard picture of computer architectures, with a number of system components connected by a single shared data path (figure 2.3). The key to making bus-based architectures work efficiently is to ensure contention for the bus itself is reduced by making effective use of the CPU caches.

Caches help to avoid contention of the bus, but only when the number of processors is small. For this reason, bus-based SMP systems tend not to rise to large numbers.

2.3 Crossbars

A refinement of the bus architecture, designed to eliminate the possible bottlenecks of the single shared path, is the *crossbar* interconnect, basically a series of single buses arranged to provide multiple paths between CPUs and memory (figure 2.4 on the facing page).

The use of multiple buses in a two (or even three) dimensional network means each processor can have a unique path of access to any part of the memory, thus vastly reducing the potential for bandwidth contention. This also means that the additional bandwidth can be soaked right back up by attaching more processors than a bus-based system – typical processor numbers range up to 32 or 64.

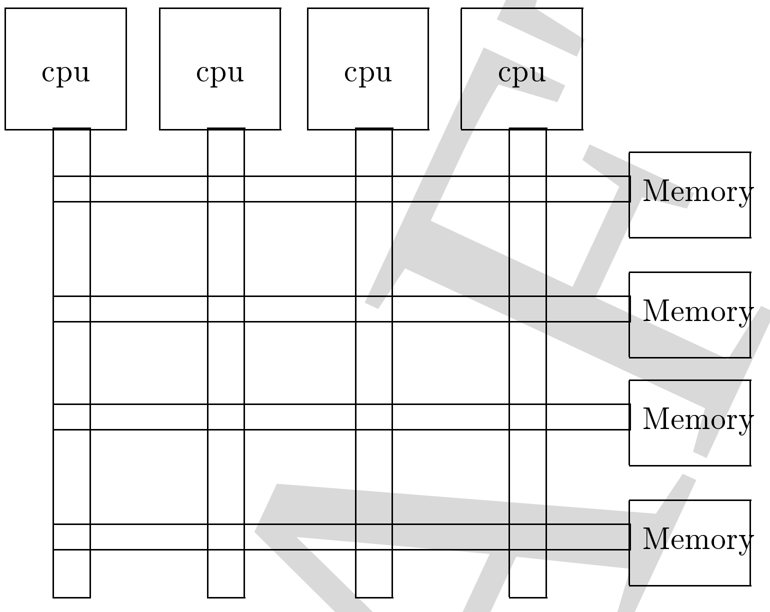

Figure 2.4: SMP crossbar

More buses, and additional support hardware, naturally means a higher price, and crossbar switches, while becoming more widespread, are still expensive and can contribute a large amount to the cost of a given HPC system.

The concept of crossbar switches is not confined to SMP systems. Similar multi-path switched interconnects are increasingly being found between SMP nodes, in ccNUMA systems and true network clusters.

2.4 ccNUMA

The rise in popularity of the single memory image offered by shared-memory SMP systems coupled with the desire to scale systems beyond the limits of bus or crossbar based systems led to the development of *distributed shared memory*.

In this model of interconnect, memory is physically distributed at the hardware level, but the system still presents a single memory image to the user. These clusters can be built in many ways, from the tightly-coupled form of *cache-coherent non-uniform memory access* (*ccNUMA*) systems to the "true cluster" form, with individual SMP boxes linked by fast network interconnects (figure 2.5 on the next page).

Naturally, processors will access "their" memory much more quickly than that of a neighbour, thus giving rise to the term *non-uniform memory access* (*NUMA*). This architecture presents problems in ensuring that the different processors' caches are maintained in a coherent state, requiring additional hardware logic.

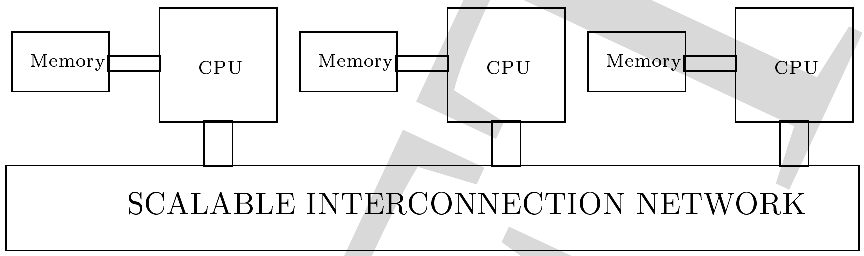

Figure 2.5: ccNUMA distributed shared-memory architecture

The distributed shared-memory ccNUMA interconnect offers the convenience of the single-memory image coupled with the large scalability of the massively parallel processing system.

The *Origin 2000* is an example of ccNUMA. In this machine each processing node contains 2 processors, a portion of shared memory, a directory for cache coherence, and two interfaces: one that connects to other I/O devices in the system and another linking additional system node boards (figure 2.6 on the facing page). A closer view of the "scalable interconnection network" rectangle in figure 2.5, called "*cray link interconnect*" for this machine, is also shown in figure 2.6 on the facing page. The network topology is a *hypercube*, since the hypercube guarantees "scalability". It has a low *diameter*: the maximum distance to traverse (number of links) between two arbytrary nodes of the 2^n nodes existing in a hypercube of dimension n is n. It also has a high *bisection bandwidth*. The bisection bandwidth of a network is the minimum number of links you have to delete to divide the network in two halves of roughly the same size. The bisection bandwidth is important, since many parallel algorithms involve massive communication between two halves of the machine. To illustrate the idea, if you have a big city divided in two parts of approximately the same size by a river and there are two bridges between the two parts, you can say the "bisection bandwitdh of the city" is two. You can imagine the jam traffic on each bridge every work day!. Is a short bisection bandwidth for such city. For the same reason, it is important to have a network with a high bisection bandwidth. the bisection bandwidth of a hypercube of dimension n is n.

In the particular case of the Origin 2000 two nodes are connected to a single router. Each router has six pairs of unidirectional links. In figure 2.6 on the next page the routers are labeled `R` and the nodes are the block boxes. Since each node has two processors, the left cube corresponds to a 32 processor machine and the one on the right to a 64 processors machine. In the 32 processor configuration, the otherwise unused router ports are shown as dotted lines being used to connect the corners of the cube, reducing the *latency* and increasing bisection bandwidth.

The message latency L is the maximum time taken for the system to deliver the first packet unit of a message. It is related with the network diameter δ and the per link latency λ. The smaller the diameter, the smaller the latency. In the abscense of other messages, the latency can be approached by $L \simeq \delta \times \lambda$.

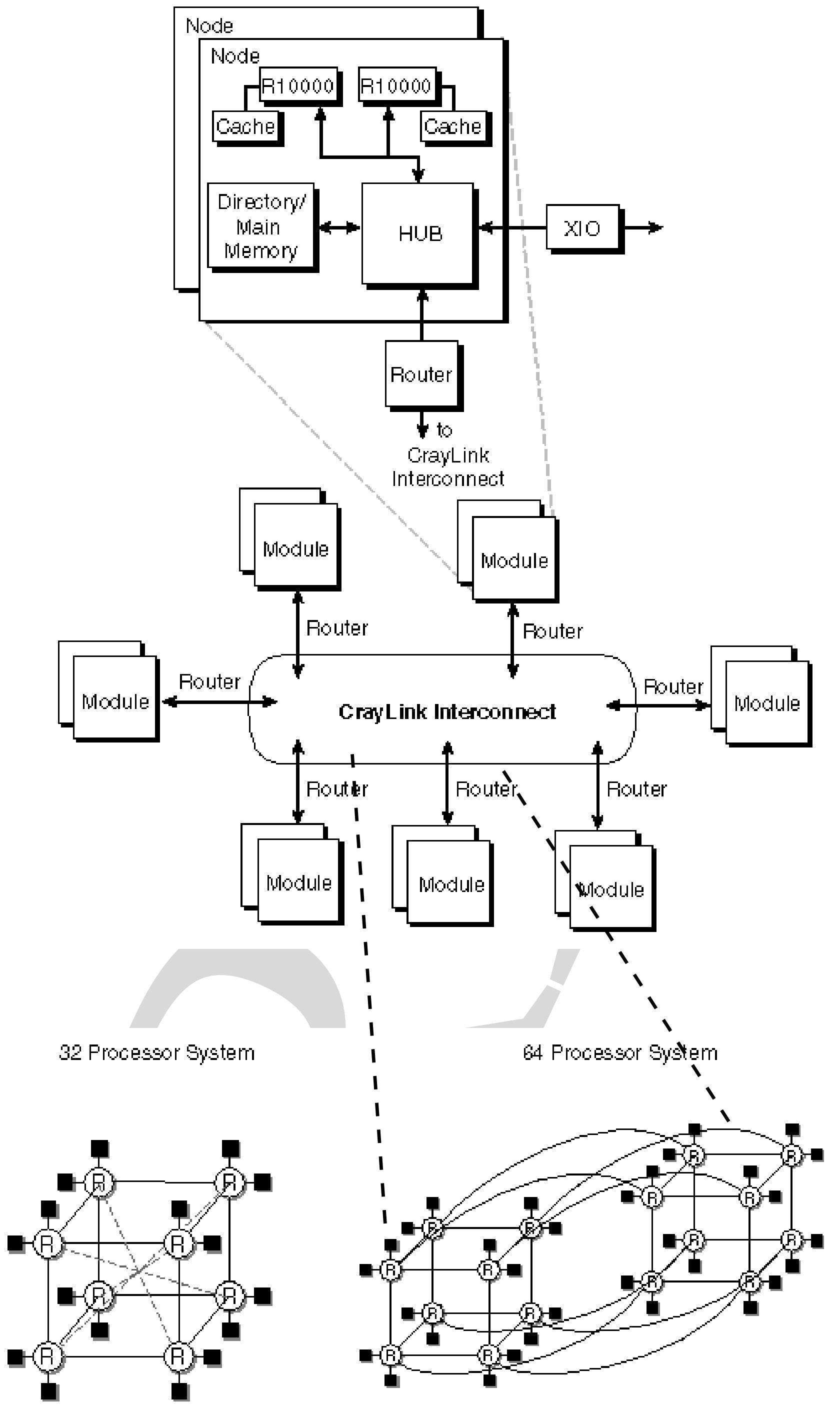

Figure 2.6: Origin 2000 Cray Link interconnection

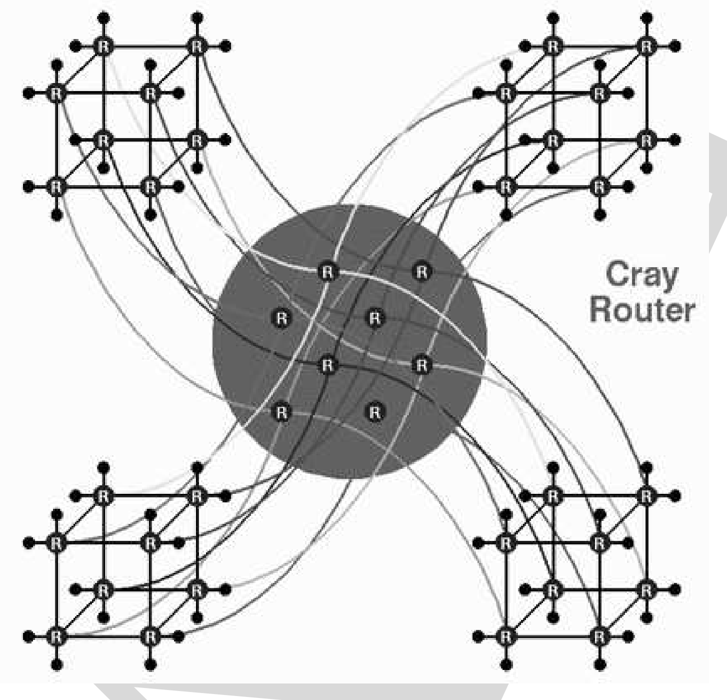

Figure 2.7: Cray Router 128 nodes

Beyond 64 processors, a hierarchical fat hypercube is employed. The eight vertices of a 32 processor hypercube are connected through their sixth link to eight metarouters. Each metarouter has 4 links. Therefore 4 32-hypercubes can be connected in this way (see figure 2.7).

To scale up to $1024 = 2^{10} = 2^5 \times 2^5$ processors, we can substitute each of the metarouters in the 128 system for a 5-D hypercube. This way we have 32 clusters of 32 processors.

2.5 Routing

Another feature of the Origin 2000 network is that it uses *wormhole routing*. Worm hole routing is a particular case of a routing technique called *cut-trough*. In cut-thorugh routing the message is divided in packets called *flits* or flow-control digits. These packets are forwarded as soon as they arrive to the router.

The flits are routed exploiting pipeline parallelism. In worm hole routing all the flits follow the same path. If, in its traverse a message needs a link that is busy, the message is blocked. To avoid this, adaptative routing techniques taking in consideration the network traffic can be used. In fact, the Origin 2000 routing system is an adaptative wormhole routing.

2.6 Cache Coherence: an Introduction

Multiprocessor systems whose CPUs are cache-based work more efficiently when their CPUs can be certain that the values they take from their caches are valid. That is, those cache values reflect the most recent values assigned to individual variables.

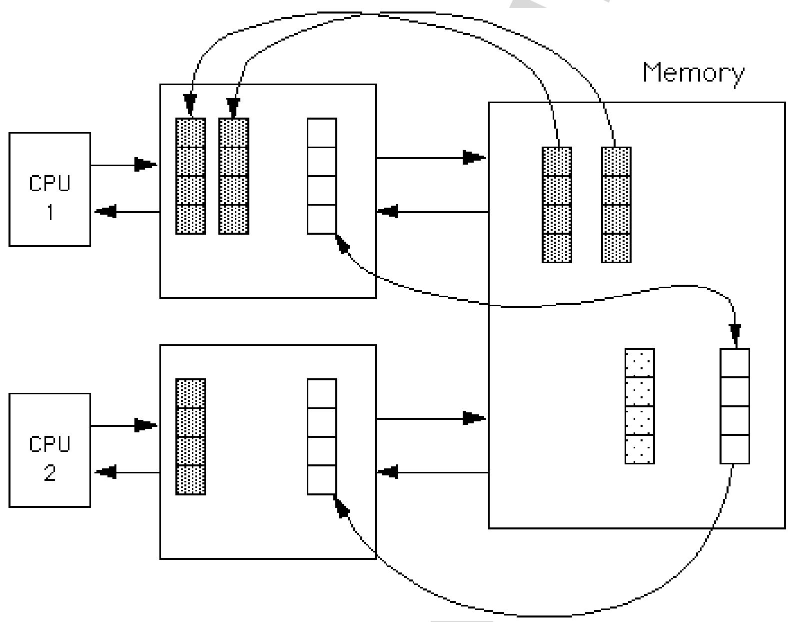

Figure 2.8: Cache coherence

Consider the case represented in figure 2.8. Both CPUs have moved the same memory area into their caches. If both CPUs do nothing but read values from that cacheline, they can be assured that they hold the most recent values. However, if one CPUs wishes to change a value in the cacheline, then the other CPU should perform no further calculations using that cacheline until it is updated. This constraint is called *cache coherence*. Cache coherence is the problem of maintaining the last correct value of data in all the caches. It can be enforced in a variety of ways. For example:

- Some bus-based multiprocessor architectures listen to all memory access activity (*bus snooping*) on the single bus shared among the processors, so that they can tell which processors are using which cachelines, and use this information to coordinate changes. Since the Origin is not bus-based this method is not possible.
- The writer broadcast the new value to all the caches holding copies of the data. This is called *update based protocol*.
- The writer asks for exclusive access to the cache line. All other caches have to invalidate their entries. Once a processor has exclusive access, it can continue to write the line as often as it likes. If another process tries

to read the line, it has to ask the writer to give up exclusive access and to convert the line back to shared state. This is called *invalidation based protocol*.

2.7 Directory Based Cache Coherency

Since the Origin has no central bus, there is no way for a CPU in one node to know when a CPU in a different node modifies its own memory or memory in yet a third node. Origin systems could mimic what is done in bus-based systems; that is, all memory accesses could be broadcast to all nodes so that stale cache lines could be invalidated. But since all nodes would need to broadcast their memory accesses, this coherency traffic would grow as the square of the number of nodes. As more nodes are added to the system, all the available internode bandwidth would eventually be consumed by this traffic. As a result, the system would suffer from limitations similar to bus-based computers: it would not be scalable. So to permit the system to scale to large numbers of processors, Origin uses what is called a *directory-based cache coherency*.

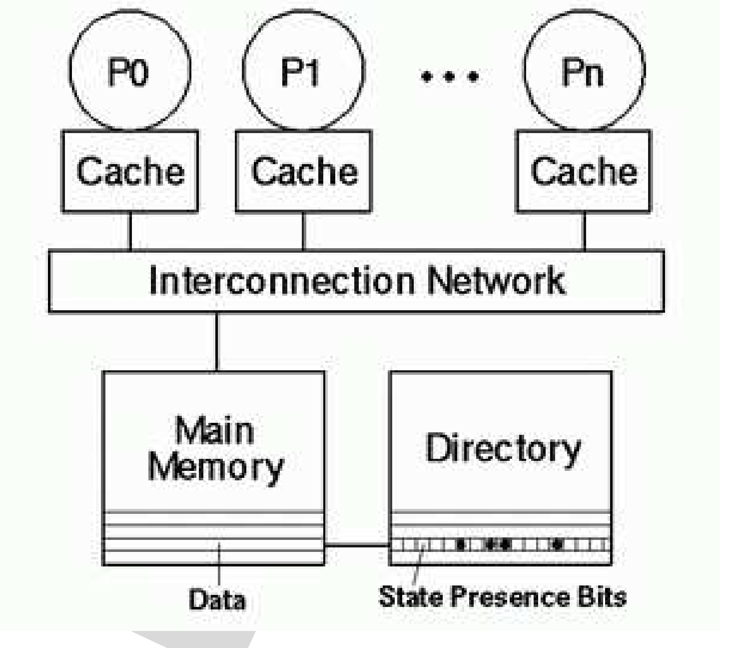

Figure 2.9: Directory based cache coherency

Memory is organized by cache lines of 128 bytes. As shown in figure 2.9, associated with the data bits in each cache line is an extra set of bits, the state presence bits — one bit per node (in large systems, those with more than 128 processors, one state presence bit represents more than one node, but the principle is the same), and a single number, the number of a node that owns the line exclusively. Whenever a node requests a cache line, its hub initiates the memory fetch of the whole line from the node that contains that line. When the cache line is not owned exclusively, the line is fetched and the state presence bit for the calling node is set. As many bits can be set as there are nodes in the system.

When a CPU wants to modify a cache line, it must gain exclusive ownership. To do so, the hub retrieves the state presence bits for the target line and sends an invalidation event to each node that has made a copy of the line. Typically, invalidations need to be sent to only a small number of other nodes; thus, the coherency traffic only grows proportionally to the number of nodes, and there is sufficient bandwidth to allow the system to scale. Both CPUs on each node sent an invalidation request discard their cached copy of the line. The number of the updating node is set in the directory data for that line as the exclusive owner. When a CPU no longer needs a cache line (for example, when it wants to reuse the cache space for other data), the hub gives up exclusive access, if it has it, and clears its state presence bit.

When a CPU wants to read a cache line and the line is exclusively owned, the hub requests a copy of the line from the owning node. This retrieves the latest copy without waiting for it to be written back to memory. There are also protocols by which CPUs can exchange exclusive control of a line when both are trying to update it and protocols that allow kernel software to invalidate all cached copies of a range of addresses.

The directory-based cache coherency mechanism uses a lot of dedicated circuitry in the hub to ensure that many CPUs can use the same memory, without race conditions, at high bandwidth. As long as memory reads far exceed memory writes, there is no performance cost for maintaining coherence.

However, when two or more CPUs alternately and repeatedly update the same cache line, performance suffers because every time either CPU refers to that memory, a copy of the cache line must be obtained from the other CPU. This performance problem is generically referred to as *cache coherency contention*. Two variations of it are:

- *Memory contention*, in which two (or more) CPUs try to update the same variables.

- *False sharing*, in which the CPUs update distinct variables that only coincidentally occupy the same cache line.

Memory contention occurs because of the design of the algorithm; correcting it generally involves an algorithmic change. False sharing is contention that arises by coincidence, not design; it can usually be corrected by modifying data structures. Part of performance tuning of parallel programs is recognizing cache coherency contention and eliminating it.

Chapter 3

Performance

Profiling is the set of methods to analyze your code in order to improve its performance. The analysis at this level is concerned with higher-level algorithm issues. The compiled code is already as optimized as the compiler can make it; you are now looking for ways to refine the program algorithm itself to make it execute more efficiently.

A set of question to answer could be this:

- What can I change in my program to improve its performance?
- What resources is my program using?

  For example, how much CPU time is my program using?
- Where in my program are most of these resources being used?

  For example, is there a particular function where my program is spending most of its time? Which line or instruction in the function accounts for most of the time being spent?
- How did the program arrive at that line or instruction in the execution process?

The term *profiling* refers to the set of techniques and tools that allow us to know the spots where a program spent its time, which functions called wich other functions during execution, what are the "bottlenecks" and what parts of your program deserve the efforts of being optimized. A program that helps you to "profile" your program is known as a "profiler".

Most profilers work as follows: when an event exceeds a threshold, a signal is delivered. On systems that do not support counter overflow at the operating system level, this signal is `SIGPROF`. Other systems allow the user to specify that an arbitrary signal is delivered. Part of the arguments to the `SIGPROF` signal handler are the interrupted process's call stack and register set. The register set contains the address at which the process was interrupted when the signal was delivered. Performance tools like Unix `prof` extract this address and hash the value into a histogram. At program completion, this histogram is analyzed and associated with symbolic information contained in the executable. What results is a line-by-line account of where counter overflows occurred in the program. GNU `gprof` in conjunction with the `-p` option of `gcc` performs exactly this analysis using process virtual time as the overflow trigger.

3.1 PAPI: A Portable Performance API

If you "profile" by hand, the first step is to measure the use of the resources in your machine: time, memory, registers, cache memories, etc. To help you with this, there is a libray called PAPI ([3]). Most of nowadays processors have hardware counters to provide information about the events that have an important impact on the performance. The *Performance API* (*PAPI*, [3]) is a portable library that provides functions to gain access to the hardware counters.

PAPI provides two interfaces to the underlying counter hardware:

1. A simple, high level interface for the acquisition of simple measurements and

2. A fully programmable, low level interface directed towards users with more sophisticated needs.

The high level interface simply provides the ability to start, stop and read the counters for a specified list of events. The low level interface deals with hardware events in groups called *EventSets*.

EventSets can reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level 1 cache misses can help detect poor locality and memory management.

The topmost layer of PAPI consists of the the high and low level PAPI interfaces. This layer is completely machine independent. It contains all of the API functions. All data structure allocation, management and state handling is done by this layer exclusively. In addition, this layer contains code to provide some more advanced functionality not always provided by the operating system, namely event multiplexing and overflow handling. This portable code calls the substrate, the internal PAPI layer that handles the machine dependent portions of accessing the counters (figure **??** on page ??).

EventSets are managed by the user through the use of integer handles. The user is free to allocate and use any number of them provided the substrate can supply the required resources. They may be used simultaneously and in fact may even share counter values.

If the user tries to add more events to an EventSet than are simultaneously countable on the platform in use, PAPI will return an appropriate error code, unless the user has enabled multiplexing. The same holds true if the user starts additional Event Sets that require counter hardware currently occupied by another running EventSet. Most modern microprocessors have a limited number of events that can be counted simultaneously. This restricts the amount of information the user can gather about a code in one pass. As a result, large applications with many hours of run time may require days or weeks of profiling in order for the developer to gather enough information from which to base an analysis. This limitation can be alleviated by multiplexing the counter hardware, or subdividing its usage over the time domain. This functionality presents the developer with the view that any set of hardware events are countable all the time. Naturally, multiplexing introduces some overhead and has the potential to adversely affect the accuracy of reported counter values.

`PAPI_QUIET`	Quietly handle errors
`PAPI_VERB_ECONT`	Print error message and continue
`PAPI_VERB_ESTOP`	Print error message and exit

Table 3.1: PAPI debug level constants

3.2 Measuring events in sequential programs with PAPI

We will use the program listed in 3.1 to introduce the most elemental features of PAPI. The function `int PAPI_library_init(int version)` initializes the PAPI library. It must be called before any PAPI functions can be used. Upon initialization, PAPI checks the version argument against the value of `PAPI_VER_CURRENT` when the library was compiled. This guards against portability problems when updating the PAPI shared libraries on your system. On success, this function returns `PAPI_VER_CURRENT`. On error, a non-zero error code is returned.

Listing 3.1 zero.c: initialization

```
1  #include "papiStdEventDefs.h"
2  #include "papi.h"
3  #include "papi_internal.h"
4  #include "test_utils.h"
5
6  int main() {
7    int retval, num_tests = 1, tmp;
8    int EventSet1;
9    int mask1 = 0x5;
10   int num_events1;
11   long long **values;
12   long long elapsed_us, elapsed_cyc;
13
14   retval = PAPI_library_init(PAPI_VER_CURRENT);
15   if (retval != PAPI_VER_CURRENT) exit(1);
16   if (PAPI_set_debug(PAPI_VERB_ECONT) != PAPI_OK) exit(1);
17   EventSet1 = add_test_events(&num_events1,&mask1);
18   values = allocate_test_space(num_tests, num_events1);
```

The function `int PAPI_set_debug(int debuglevel)` sets the default debug level for the PAPI library. Here `debuglevel` is one of the constants defined in table 3.1. The current debug level is internally stored in the PAPI library and is used by the default internal PAPI error handler subroutine. The error handler is called by library routines on the occurence of recoverable errors.

The function `int add_test_events(int *number, int *mask)` called in line 17 of listing 3.1 is implemented in file `test_utils.c`. Its code is sketched in listing 3.2 on the next page.

The call to function `int PAPI_get_opt(int option, PAPI_option_t *ptr)` at line 6 of listing 3.2 on the following page query the options of the PAPI li-

Listing 3.2 File `test_utils.c`. Function `add_test_events`

```
int add_test_events(int *number, int *mask) {
  int retval;
  int EventSet = PAPI_NULL;

  *number = 0;
  retval = PAPI_get_opt(PAPI_GET_MAX_HWCTRS,NULL);
  if (retval < 1) exit(1);

  if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    exit(1);
  ...
  if (*mask & 0x4) {
      retval = PAPI_add_event(&EventSet, PAPI_FP_INS);
      if (retval == PAPI_OK) (*number)++;
      else {
        fprintf(stderr,"PAPI_FP_INS is not available.\n");
        *mask = *mask ^ 0x4;
      }
    }
  if (*mask & 0x1) {
      retval = PAPI_add_event(&EventSet, PAPI_TOT_CYC);
      if (retval == PAPI_OK)
      (*number)++;
      else {
          fprintf(stderr, "PAPI_TOT_CYC is not available.\n");
          *mask = *mask ^ 0x1;
        }
    }
  ...
  return(EventSet);
}
```

brary. The second argument is a pointer to the `PAPI_option_t` structure. Not all options require or return information in this structure. This is the case here. The input parameter `option` describes the course of action. In this case, it allow us to know the number of hardware counters. For other legal request values see table 3.2 on the next page.

A PAPI event set is created at line 9. An event set is a data structure that contains key information about the events being considered: what low-level hardware counters to use, the most recently read counter values, the state of the event set (running/not running), the option settings (e.g., domain, granularity, overflow, profiling), etc.

The function `PAPI_create_eventset()` creates a new `EventSet` pointed to by `EventSet`, which must be previously initialized to `PAPI_NULL` (see line 3). From this point on, the user may then add hardware events to the event set by calling `PAPI_add_event`.

The function `int PAPI_add_event(int *EventSet, int Event)` has two parameters. It creates a new `EventSet` or modifies an existing one and adds the hardware counter `Event` to the `EventSet`. This function returns an error on counter resource conflicts. Thus, the call at line 13 includes the `Event PAPI_FP_INS` and the one at line 21 the `Event PAPI_TOT_CYC`. The first event allow us to count the number of floating point instructions executed. The second the total number of cycles. These are the two event we keep under study, since we have define the parameter `mask` in line 9 of listing 3.1 on page 35 as `int mask1 = 0x5 = 0x4 + 0x1`. The number of events you can watch is quite extensive. Table 3.4 on page 41 shows some of them. On a given platform, a given event can either be directly available as a single counter, derived using a combination of counters or simply unavailable.

The integer containing the `EventSet` is finally returned by the function `add_tests_events` and stored in the variable `EventSet1` in line 17 of listing 3.1 on page 35. At the same time the number of events that have been succesfully set (that is not necessarily equal to the number of requested events) is returned in `num_events1`. A matrix `values` of dimensions `num_tests` $\times$ `num_events` to keep the result of the experiments is allocated at line 18. The code of the auxiliary function `allocate_test_space()` performing the memory assignment is depicted in listing 3.3 on the next page. The code of function `main()` continues in listing 3.4 on page 39. The function `PAPI_get_real_usec()` returns a real (wall) time in micro seconds. It is used at lines 19 and 26 to calculate the real time in micro seconds for the section between these two lines. Analogously, the pair of calls to function `PAPI_get_real_cyc()` at lines 20 and 27 is used to calculate the number of total cycles elapsed while the section is executed. That information, printed at lines 40 and 41 appear in the execution window 3.1 on page 42.

As the name of the function indicates, the call `PAPI_start(EventSet1)` at line 21 starts counting all of the hardware events contained in `EventSet1`. All the associated counters are implicitly set to zero. `PAPI_start()` will not start the hardware counters if a conflicting `EventSet` is already running and the counters are already in use. The return value is an integer that indicates whether the call succeeded (`PAPI_OK`) or failed (not `PAPI_OK`). Line 23 is the code being profiled. Function `do_flops(int n)` is a loop of size `n` performing

`PAPI_GET_CLOCKRATE`	Return clockrate in MHz.
`PAPI_GET_MAX_CPUS`	Return number of CPUs.
`PAPI_GET_MAX_HWCTRS`	Return number of counters.
`PAPI_GET_EXEINFO`	Addresses for text/data/bss.
`PAPI_GET_HWINFO`	Info. about hardware.
`PAPI_GET_PRELOAD`	Get `LD_PRELOAD` environment equivalent.

Table 3.2: Some Legal information request values for `PAPI_get_opt`

Listing 3.3 File `test_utils.c`. Function `allocate_test_space`

```
long long **allocate_test_space(int num_tests, int num_events)
{
  long long **values;
  int i;

  values = (long long **)malloc(num_tests*sizeof(long long *));
  if (values==NULL)
    exit(1);
  memset(values,0x0,num_tests*sizeof(long long *));

  for (i=0;i<num_tests;i++)
    {
      values[i] = (long long *)malloc(num_events*sizeof(long long));
      if (values[i]==NULL)
        exit(1);
      memset(values[i],0x00,num_events*sizeof(long long));
    }
  return(values);
}
```

Listing 3.4 Program zero.c: continuation

```
    elapsed_us = PAPI_get_real_usec();
    elapsed_cyc = PAPI_get_real_cyc();
    retval = PAPI_start(EventSet1);
    if (retval != PAPI_OK) exit(1);
    do_flops(NUM_FLOPS);
    retval = PAPI_stop(EventSet1, values[0]);
    if (retval != PAPI_OK) exit(1);
    elapsed_us = PAPI_get_real_usec() - elapsed_us;
    elapsed_cyc = PAPI_get_real_cyc() - elapsed_cyc;
    remove_test_events(&EventSet1, mask1);
    printf("Test case 0: start, stop.\n");
    tmp = PAPI_get_opt(PAPI_GET_DEFDOM,NULL);
    printf("Default domain is: %d (%s)\n",
            tmp,stringify_domain(tmp));
    tmp = PAPI_get_opt(PAPI_GET_DEFGRN,NULL);
    printf("Default granularity is: %d (%s)\n",
            tmp,stringify_granularity(tmp));
    printf("Using %d iterations of c += a*b\n",NUM_FLOPS);
    printf("Test type   : \t1\n");
    printf("PAPI_FP_INS : \t%lld\n", (values[0])[0]);
    printf("PAPI_TOT_CYC: \t%lld\n", (values[0])[1]);
    printf("Real usec   : \t%lld\n", elapsed_us);
    printf("Real cycles : \t%lld\n", elapsed_cyc);
    printf("Verification:\n");
    free_test_space(values, num_tests);
    PAPI_shutdown();
    exit(0);
  }
```

PAPI_DOM_USER	User context counted
PAPI_DOM_MIN	The smallest available context
PAPI_DOM_KERNEL	Kernel/OS context counted
PAPI_DOM_OTHER	Exception/transient mode
PAPI_DOM_ALL	All contexts counted
PAPI_DOM_MAX	The largest available context

Table 3.3: PAPI: Domain values

the assignment statement `c += a * b;`. The corresponding call to function `int PAPI_stop()` terminates the counting of the hardware events contained in `EventSet1`. If, like it is in this case, the pointer `values` is non-NULL, the counters contained in `EventSet1` are copied into `values`.

After that point, the events associated with `EventSet1` are no longer needed and we can free the allocated resources. The call at line 28 takes care of it. The code of the corresponding function `remove_test_events()` appears in listing 3.5 on the facing page. This function calls `PAPI_rem_event()` for each of the events in `mask1`. The interface of the last is:

`int PAPI_rem_event (int *EventSet, int Event)`.

This function frees the hardware counter in `Event` from `EventSet`. After every used counter has been removed, the call to `destroy_eventset()` at line 17 effectively removes `EventSet1` from existence. Is a requirement of this function, that `EventSet1` must be empty.

Returning to `main()`, lines 30 and 33 in listing 3.4 on the page before show the use of function `PAPI_get_opt (int option, PAPI_option_t *ptr)`. This function queries the status of tunable options. The `ptr` structure is for input and output. Not all options fill the `PAPI_option_t` structure. Domain is one of the constants in table 3.3. The default value is `PAPI_DOM_USER`. Granularity settings include per thread, process, process group, current cpu, and each cpu. By default, the granularity is set to the most restrictive supported by the substrate.

3.3 Tuning

The natural way to write a parallel algorithm for a given software-hardware platform, is to do it for a generic number of processors P and not for a specific number, say 4. In the same way, you can write your (not necessarily parallel) algorithm taking into account the size of the cache and the length of the cache line. Again, you will not write your program so that it works only with specific values but for generic ones (S, L). Obviously, if your program is correct, its output for a given input does not depend of the specific values of parameters like the number of processors, the size of the L2 cache, the cacheline length or the size of the input buffers. But these parameters may have a strong impact on the performance, that is, in the time spent by the algorithm. We say that such parameters as $P, S, L, \dots$ are *tuning parameters*.

PAPI_CA_SHR	Requests for exclusive access to shared cache line
PAPI_FLOPS	Floating point instructions per second
PAPI_FP_STAL	Cycles the FP unit
PAPI_FPU_IDL	Cycles floating point units are idle
PAPI_INT_INS	Integer instructions
PAPI_IPS	Instructions per second
PAPI_TOT_CYC	Total cycles
PAPI_L1_DCA	L1 data cache accesses
PAPI_L1_DCH	L1 data cache hits
PAPI_L2_ICH	L2 instruction cache hits
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L2_ICR	L2 instruction cache reads
PAPI_L2_ICW	L2 instruction cache writes
PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses
PAPI_L2_TCH	L2 total cache hits
PAPI_L2_TCM	Level 2 cache misses
PAPI_L2_TCR	L2 total cache reads
PAPI_L2_TCW	L2 total cache writes
PAPI_LD_INS	Load instructions

Table 3.4: Some PAPI events

Listing 3.5 File `test_utils.c`: Function `remove_test_events`

```
int remove_test_events(int *EventSet, int mask)
{
  int retval = PAPI_OK;

  if (mask & 0x400)
    {
      retval = PAPI_rem_event(EventSet, PAPI_L2_TCH);
      if (retval < PAPI_OK) return(retval);
    }
  ...
  if (mask & 0x1)
    {
      retval = PAPI_rem_event(EventSet, PAPI_TOT_CYC);
      if (retval < PAPI_OK) return(retval);
    }

  return(PAPI_destroy_eventset(EventSet));
}
```

```
nereida:tests$make zero
cc  -I..  -c do_loops.c
cc  -I..  -c test_utils.c
cc  -I..  zero.c do_loops.o test_utils.o -lpapi  -o zero
nereida:tests$zero
Test case 0: start, stop.
-----------------------------------------------
Default domain is: 1 (PAPI_DOM_USER)
Default granularity is: 1 (PAPI_GRN_THR)
Using 10000000 iterations of c += a*b
-------------------------------------------------
Test type   :   1
PAPI_FP_INS :   20000000
PAPI_TOT_CYC:   83183582
Real usec   :   142596
Real cycles :   86269034
-------------------------------------------------
Verification:
nereida:tests$
```

Execution 3.1: The execution of program zero

In general, what we call a *Tunable Algorithm* is a family of algorithms $(A)_{t\in T}$ indexed in a set of tuning parameters T holding the *tuning condition.* The tuning condition implies that

$$A_{t_0}(x) = A_{t_1}(x)\ \forall \text{ tuning parameters } t_0, t_1 \in T \text{ and } \forall \text{ input } x \tag{3.1}$$

The tuning problem is to find the optimal values of the tuning parameters of a given algorithm. That is, to find t_0 such that

$$Time(A_{t_0}(x)) \leq Time(A_t(x))\ \forall t \in T \text{ and } \forall \text{ input } x \tag{3.2}$$

Where $Time(A_t(x))$ is the time spent executing the t-th algorithm A_t on input x.

Unfortunately, not always the problem stated in equation 3.3 has a solution. It often happens that the solution t_0 depends on the input x and therefore, we have to face the most *general tuning problem* of finding a function $t_0(x)$ such that:

$$Time(A_{t_0(x)}(x)) \leq Time(A_t(x))\ \forall t \in T \text{ and } \forall \text{ input } x \tag{3.3}$$

3.3.1 Example

Let us see an example of tuning. To fix ideas and the family of tuning parameters, let us introduce first the (Z, L) ideal-cache model [9].

This model consists of a computer with a two level memory hierarchy consisting of an ideal data cache of Z words and a arbitrarily large main memory. The model assumes that the word size is constant. As usual, the cache is partitioned in cache lines, each consisting of $L > 1$ consecutive words that are always moved together between cache and main memory. The model assumes also a *tall cache*:

$$Z = \Omega(L^2) \tag{3.4}$$

In fact, we can assume that Z is much larger than L^2. The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a cache hit ocuurs, and the word is delivered to the processor. Otherwise, a cache miss occurs, and the line is fetched into the cache. The ideal cache is *fully associative*: cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line strategy of replacing the cache line whose next access is farthest in the future, and thus it exploits temporal locality perfectly.

Consider the problem of multiplying two $n \times n$ matrices A and B to produce their $n \times n$ product C. Assume n is large. The conventional approach to design a *cache aware* matrix product is to divide the matrices in blocks. Each matrix is viewed as consisting of $\frac{n}{s} \times \frac{n}{s}$ submatrices M_{ij} (the blocks), each of which has size $s \times s$, where $s \in T$ is the "tuning parameter". The algorithm is the following:

```
BLOCK_MULT(A, B, C,n)
  for(i = 0; i < n/s; i++)
    for(j = 0; j < n/s; j++)
      for(k = 0; k < n/s; k++)
        ORD_MULT(A[i][k], B[k][j], C[i][j], s)
```

where `ORD_MULT(A, B, C, s)` is a function that computes `C = C + AB` on $s \times s$ matrices using the ordinary $O(s^3)$ algorithm.

Depending on the cache size of the machine on which `BLOCK_MULT` is run, the parameter s can be tuned to make the algorithm run fast. To solve equation 3.3 we can choose s so that the three matrices `A[i][k]`, `B[k][j]` and `C[i][j]` simultaneously fit in cache.

To store a $s \times s$ submatrix takes approximately (s^2/L) cache lines. Since we need space for the three, $3 \times \frac{s^2}{L} = Z$, we can see that s is approximately $\sqrt{\frac{ZL}{3}}$.

Thus, each call to `ORD_MULT(A[i][k], B[k][j], C[i][j], s)` runs with at most $3 \times (s^2/L)$ cache misses, needed to bring the three matrices into the cache.

3.3.1 Excercise

Implement the block matrix product algorithm, use PAPI to measure the data cache misses and compare your experimental optimal values of s with those predicted by the theoretical study.

3.4 OpenMP and PAPI

From the perspective of the OpenMP programmer, fully thread aware performance is needed. There are two specific problems.

As with any API, this interface must be *re-entrant*, because any number of threads can simultaneously call the same PAPI function. This means that any writeable global structures must be locked while in use. This has the potential of increasing overhead and introducing large sections of machine dependent code to the top layer.

Fortunately, PAPI has only one large globally writeable structure, which keeps track of the *EventSets* currently defined by the user. In addition, it is only writeable by two API calls that are almost exclusively used during the setup and shutdown of the PAPI library. The second problem is the accuracy of event counts as returned by threads calling the PAPI.

In order to support threaded operation, the operating system must save and restore the counter hardware upon context switches among different threads or processes. However, OpenMP hides the concept of user and kernel level threads from the user. As a result, unless the user explicitly takes action to bind their thread to a *kernel thread* (sometimes called a *Light Weight Process* or LWP), the counts returned by PAPI will not necessarily be accurate.

To address this situation, PAPI treats every platform as if it is running on top of kernel threads.

Unbound, user level threads that call PAPI will function properly, but will most likely return unreliable or inaccurate event counts.

Fortunately, in the batch environments of the HPC community, there is no significant advantage to user level threads and thus kernel level threads are the default.

Program `zero_omp.c` in listings 3.6 and 3.7 teaches an strategy to use PAPI from an OpenMP program. The master serial thread (listing 3.6 on page 46) executes the following steps:

1. Initialize PAPI library (lines 58-64 in listing 3.6). In case of error, the function `int PAPI_perror(int code, char *destination, int length)` is called (line 60). This function copies `length` chars of the error description string corresponding to `code` into the `destination` address. The resulting string is always null terminated.

2. The debug level is set in line 66 to `PAPI_VERB_ECONT` so that errors will be printed but the program will still continue.

3. The function `int PAPI_thread_init()` is called at lines 69-71. The interface of this function is:

   `int PAPI_thread_init(unsigned long int (*handle)(),int flag);`

   This function initializes thread support in the PAPI library. It should be called only once, just after `PAPI_library_init`, and before any other PAPI calls. Applications that make no use of threads do not need to call this routine. The argument `handle` is a pointer to a function that returns current thread `ID`. The `flag` argument is reserved for future use and should be set to zero. In case of error the corresponding message is issued at lines 72-77.

4. Line 77 sets the counter to get the μ-seconds. Analogously, line 78 sets the counter to get the cycles.

5. After saving in `maxthr` the maximum number of threads, the parallel for loop is executed (lines 84-85). The diferent iterations are shared among the threads. Since the number of iterations is equal to the number of available

threads, each iteration correspond to one thread. Hopefully, each thread will correspond to one processor.

6. Finally, the number of μ-seconds and cycles are computed (lines 87 and 88) and their values printed.

Each thread inside the Thread routine (listing 3.7 on page 47) perform the following steps:

1. Initializes the event set `EventSet1` and allocates the matrix `values` to save the results of the experiments (lines 12-13).
2. Line 15 sets the counter to get the μ-seconds. Analogously, line 16 sets the counter to get the cycles.
3. Counters are started at line 18.
4. The code to be measured is executed at line 26.
5. The counters are stopped at line 28.
6. Finally, the number of μ-seconds and cycles are computed (lines 36-37), the event set is removed (line 39), results are printed (lines 41-45) and the space allocated for values is disposed (line 47).

Listing 3.6 File zero_omp.c: Function main()

```
int main()
{
  int i, maxthr, retval;
  long long elapsed_us, elapsed_cyc;
  char errstring[PAPI_MAX_STR_LEN];

  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT){
    PAPI_perror(retval,errstring,PAPI_MAX_STR_LEN);
    printf("%s:%d::PAPI_library_init failed. %d %s\n",
           __FILE__,__LINE__,retval,errstring);
    exit(1);
  }

  if (PAPI_set_debug(PAPI_VERB_ECONT) != PAPI_OK)
    exit(1);

  retval = PAPI_thread_init(
            (unsigned long (*)(void))(omp_get_thread_num),0
           );
  if (retval != PAPI_OK){
    PAPI_perror(retval,errstring,PAPI_MAX_STR_LEN);
    printf("%s:%d::PAPI_thread_init failed. %d %s\n",
           __FILE__,__LINE__,retval,errstring);
    exit(1);
  }
  elapsed_us = PAPI_get_real_usec();
  elapsed_cyc = PAPI_get_real_cyc();

  maxthr = omp_get_num_procs();

#pragma omp parallel for
  for (i=1;i<maxthr+1;i++)
    Thread(1000000*i);

  elapsed_cyc = PAPI_get_real_cyc() - elapsed_cyc;
  elapsed_us = PAPI_get_real_usec() - elapsed_us;

  printf("Master real usec   : \t%lld\n", elapsed_us);
  printf("Master real cycles : \t%lld\n", elapsed_cyc);

  exit(0);
}
```

Listing 3.7 File zero_omp.c. Function thread().

```
...
#include "papiStdEventDefs.h"
#include "papi.h"
#include "papi_internal.h"
#include "test_utils.h"
#include <omp.h>

void Thread(int n)
{
  ... /* declarations */

  EventSet1 = add_test_events(&num_events1,&mask1);
  values = allocate_test_space(num_tests, num_events1);

  elapsed_us = PAPI_get_real_usec();
  elapsed_cyc = PAPI_get_real_cyc();

  retval = PAPI_start(EventSet1);
  if (retval != PAPI_OK) {
    tmp=PAPI_MAX_STR_LEN;
    PAPI_perror(retval,errstring,tmp);
    printf("PAPI_start failed. %d %s\n",retval,errstring);
    exit(1);
    }

  do_flops(n);

  retval = PAPI_stop(EventSet1, values[0]);
  if (retval != PAPI_OK){
    tmp=PAPI_MAX_STR_LEN;
    PAPI_perror(retval,errstring,tmp);
    printf("PAPI_stop failed. %d %s\n",retval,errstring);
    exit(1);
    }

  elapsed_us = PAPI_get_real_usec() - elapsed_us;
  elapsed_cyc = PAPI_get_real_cyc() - elapsed_cyc;

  remove_test_events(&EventSet1, mask1);

  printf("Thread 0x%x PAPI_FP_INS : \t%lld\n",
    omp_get_thread_num(), (values[0])[0]);
  printf("Thread 0x%x PAPI_TOT_CYC: \t%lld\n",
    omp_get_thread_num(), (values[0])[1]);
  ... /* more prints */

  free_test_space(values, num_tests);
}
```

Appendix A

Programs

A.1 Generic Matrix Product and All Shortest Paths

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

/* N has to be a power of two */
#define N 4
#define MAX_DIMENSION 32

#define classic

#ifdef classic
/***** the add mul object *********/
#define m_type double
#define m_neutral 0
#define m_plus(a,b) ((a)+(b))
#define m_times(a,b) ((a)*(b))
#define m_format_in "%lf"
#define m_format_out "%lf\t"
/****** end add mul object ********/
#else
/***** the min add ********************/
#define m_type unsigned
#define m_neutral ULONG_MAX
#define m_plus(a,b) (((a)<(b))?(a):(b))
#define m_times(a,b) ((a)+(b))
#define m_format_in "%d"
#define m_format_out "%d\t"
/***** end min add object ***********/
#endif

typedef m_type matrix[N][N];

matrix a;
matrix temp;
```

```
matrix c;

void printMat(matrix c) {
  int i,j;

  for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
      printf(m_format_out, c[i][j]);
    }
    printf("\n");
  }
}

void m_init() {
  int i, j;
  FILE * file;

  file = fopen("data","r");

  for(i=0;i<N;i++)
    for(j=0;j<N;j++) {
      fscanf(file,m_format_in,&a[i][j]);
    }
}

void m_mat(matrix c, matrix a, matrix b) {
  int i, j, k;
  double sum;

#pragma omp parallel
#pragma omp for private(i, j, k, sum)
  for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
      sum = m_neutral;
      for(k = 0; k < N; k++)
        sum = m_plus(sum, m_times(a[i][k], b[k][j]));
      c[i][j] = sum;
    }
  }
}

void m_assign(matrix a, matrix b) {
  int i,j;

  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
      a[i][j] = b[i][j];
}

void m_power(matrix c, matrix a, int k) {

  m_assign(temp, a);
  m_mat(c, temp, temp);
  k /=2;
```

```
  while(k > 1) {
    m_assign(temp, c);
    m_mat(c, temp, temp);
    k /=2;
  }
}

int main() {
  int i, j, k;


    m_init();
    printf("A\n");
    printMat(a);

    m_power(c, a, N);

    printf("C\n");
    printMat(c);
}
```

A.2 PAPI

A.2.1 Makefile

```
CC_R    = ompcc
OMP = zero_omp
SMP = zero_smp
SHMEM = zero_shmem
SERIAL  = zero misses locality locality2 cache_pp
ALL  = $(OMP) $(SERIAL)
UTILOBJS= do_loops.o test_utils.o
INCLUDE = -I..

omp: $(OMP)

smp: $(SMP)

pthreads: $(PTHREADS)

shmem: $(SHMEM)

serial: $(SERIAL)

zero_omp: zero_omp.c $(UTILOBJS)
-$(CC_R) $(OMPCFLGS) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) zero_omp.c $(UTILOBJS) -lpapi $(LDFLAGS) -o zero_omp $(OMPLIBS)

cache_pp: cache_pp.c $(UTILOBJS)
-$(CC_R) $(OMPCFLGS) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) cache_pp.c $(UTILOBJS) -lpapi $(LDFLAGS) -o cache_pp $(OMPLIBS)

zero: zero.c $(UTILOBJS)
$(CC) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) zero.c $(UTILOBJS) -lpapi $(LDFLAGS) -o zero

locality: locality.c $(UTILOBJS)
$(CC) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) locality.c $(UTILOBJS) -lpapi $(LDFLAGS) -o locality

locality2: locality2.c $(UTILOBJS)
$(CC) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) locality2.c $(UTILOBJS) -lpapi $(LDFLAGS) -o locality2

misses: misses.c $(UTILOBJS)
$(CC) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) misses.c $(UTILOBJS) -lpapi $(LDFLAGS) -o misses

misses2: misses2.c $(UTILOBJS)
$(CC) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) misses2.c $(UTILOBJS) -lpapi $(LDFLAGS) -o misses2

do_loops.o: do_loops.c do_loops.h
$(CC) $(CFLAGS) $(INCLUDE) $(NOOPT) -c do_loops.c

test_utils.o: test_utils.c
$(CC) $(CFLAGS) $(INCLUDE) $(OPTFLAGS) -c test_utils.c

clean:
rm -f *.o $(ALL) core *~
rm -rf rii_files *.stderr *.stdout
```

A.2.2 test_utils.c

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#undef NDEBUG
#include "papiStdEventDefs.h"
#include "papi.h"
#include "papi_internal.h"
#include "test_utils.h"

long long **allocate_test_space(int num_tests, int num_events)
{
  long long **values;
  int i;

  values = (long long **)malloc(num_tests*sizeof(long long *));
  if (values==NULL)
    exit(1);
  memset(values,0x0,num_tests*sizeof(long long *));

  for (i=0;i<num_tests;i++)
    {
      values[i] = (long long *)malloc(num_events*sizeof(long long));
      if (values[i]==NULL)
exit(1);
      memset(values[i],0x00,num_events*sizeof(long long));
    }
  return(values);
}

void free_test_space(long long **values, int num_tests)
{
  int i;

  for (i=0;i<num_tests;i++)
    free(values[i]);
  free(values);
}

/* Mask tells us what to select.
   0x400 is PAPI_L2_TCH.
   0x200 is PAPI_L2_TCA.
   0x100 is PAPI_L2_TCM.
   0x40 is PAPI_L1_DCM.
   0x20 is PAPI_L1_ICM.
   0x10 is PAPI_L1_TCM.
   0x8 is PAPI_FLOPS.
   0x4 is PAPI_FP_INS.
   0x2 is PAPI_TOT_INS.
   0x1 is PAPI_TOT_CYC. */
```

```
int add_test_events(int *number, int *mask)
{
  int retval;
  int EventSet = PAPI_NULL;

  *number = 0;

  retval = PAPI_get_opt(PAPI_GET_MAX_HWCTRS,NULL);
  if (retval < 1)
    exit(1);

  if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    exit(1);

  if (*mask & 0x400)
    {
      retval = PAPI_add_event(&EventSet, PAPI_L2_TCH);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_L2_TCH is not available.\n");
  *mask = *mask ^ 0x400;
}
    }

  if (*mask & 0x200)
    {
      retval = PAPI_add_event(&EventSet, PAPI_L2_TCA);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_L2_TCA is not available.\n");
  *mask = *mask ^ 0x200;
}
    }

  if (*mask & 0x100)
    {
      retval = PAPI_add_event(&EventSet, PAPI_L2_TCM);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_L2_TCM is not available.\n");
  *mask = *mask ^ 0x100;
}
    }

  if (*mask & 0x40)
    {
```

```
      retval = PAPI_add_event(&EventSet, PAPI_L1_DCM);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_L1_DCM is not available.\n");
  *mask = *mask ^ 0x40;
}
    }

  if (*mask & 0x20)
    {
      retval = PAPI_add_event(&EventSet, PAPI_L1_ICM);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_L1_ICM is not available.\n");
  *mask = *mask ^ 0x20;
}
    }

  if (*mask & 0x10)
    {
      retval = PAPI_add_event(&EventSet, PAPI_L1_TCM);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_L1_TCM is not available.\n");
  *mask = *mask ^ 0x10;
}
    }

  if (*mask & 0x8)
    {
      retval = PAPI_add_event(&EventSet, PAPI_FLOPS);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_FLOPS is not available.\n");
  *mask = *mask ^ 0x8;
}
    }

  if (*mask & 0x4)
    {
#if defined(__digital__)
      fprintf(stderr,"Using PAPI_TOT_INS instead of PAPI_FP_INS.\n");
      retval = PAPI_add_event(&EventSet, PAPI_TOT_INS);
#else
      retval = PAPI_add_event(&EventSet, PAPI_FP_INS);
#endif
```

```
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_FP_INS is not available.\n");
  *mask = *mask ^ 0x4;
}
    }

  if (*mask & 0x2)
    {
      retval = PAPI_add_event(&EventSet, PAPI_TOT_INS);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_TOT_INS is not available.\n");
  *mask = *mask ^ 0x2;
}
    }

  if (*mask & 0x1)
    {
      retval = PAPI_add_event(&EventSet, PAPI_TOT_CYC);
      if (retval == PAPI_OK)
(*number)++;
      else
{
  fprintf(stderr,"PAPI_TOT_CYC is not available.\n");
  *mask = *mask ^ 0x1;
}
    }

  return(EventSet);
}

int remove_test_events(int *EventSet, int mask)
{
  int retval = PAPI_OK;

  if (mask & 0x400)
    {
      retval = PAPI_rem_event(EventSet, PAPI_L2_TCH);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x200)
    {
      retval = PAPI_rem_event(EventSet, PAPI_L2_TCA);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x100)
    {
```

```
      retval = PAPI_rem_event(EventSet, PAPI_L2_TCM);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x40)
    {
      retval = PAPI_rem_event(EventSet, PAPI_L1_DCM);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x20)
    {
      retval = PAPI_rem_event(EventSet, PAPI_L1_ICM);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x10)
    {
      retval = PAPI_rem_event(EventSet, PAPI_L1_TCM);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x8)
    {
      retval = PAPI_rem_event(EventSet, PAPI_FLOPS);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x4)
    {
      retval = PAPI_rem_event(EventSet, PAPI_FP_INS);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x2)
    {
      retval = PAPI_rem_event(EventSet, PAPI_TOT_INS);
      if (retval < PAPI_OK) return(retval);
    }

  if (mask & 0x1)
    {
      retval = PAPI_rem_event(EventSet, PAPI_TOT_CYC);
      if (retval < PAPI_OK) return(retval);
    }

  return(PAPI_destroy_eventset(EventSet));
}

char *stringify_domain(int domain)
{
  switch(domain)
    {
    case PAPI_DOM_USER:
```

```
      return("PAPI_DOM_USER");
    case PAPI_DOM_KERNEL:
      return("PAPI_DOM_KERNEL");
    case PAPI_DOM_OTHER:
      return("PAPI_DOM_OTHER");
    case PAPI_DOM_ALL:
      return("PAPI_DOM_ALL");
    default:
      abort();
    }
  return(NULL);
}

char *stringify_granularity(int granularity)
{
  switch(granularity)
    {
    case PAPI_GRN_THR:
      return("PAPI_GRN_THR");
    case PAPI_GRN_PROC:
      return("PAPI_GRN_PROC");
    case PAPI_GRN_PROCG:
      return("PAPI_GRN_PROCG");
    case PAPI_GRN_SYS_CPU:
      return("PAPI_GRN_SYS_CPU");
    case PAPI_GRN_SYS:
      return("PAPI_GRN_SYS");
    default:
      abort();
    }
  return(NULL);
}
```

A.2.3 do_loops.c

```
/* Compile me with -O0 or else you'll get none. */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include "test_utils.h"
#include "do_loops.h"

/*
#define L1_MISS_BUFFER_SIZE_INTS 128*1024
*/
#define L1_MISS_BUFFER_SIZE_INTS 1
static int buf[L1_MISS_BUFFER_SIZE_INTS];

volatile double a = 0.5, b = 6.2;
volatile long long z = -101010101;

double c[N][N];

void do_flops(int n)
{
  int i;
  double c = 0.11;

  for (i=0; i < n; i++) {
    c += a * b;
  }
}

void do_locality1()
{
  int i,j;

  for (j=0; j < N; j++)
    for (i=0; i < N; i++)
      c[i][j] = 0.0;
}

void do_locality2()
{
  int i,j;

  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      c[i][j] = 0.0;
}

void do_reads(int n)
{
  int i, j;
```

```
  static int fd = -1;
  char buf;
  long long c = 0.11;

  if (fd == -1) {
      fd = open("/dev/zero",O_RDONLY);
      if (fd == -1) {
        perror("open(/dev/zero)");
        exit(1);
      }
    }

  for (i=0; i < n; i++) {
    for (j=0; j < n; j++) {
      c += z;
    }
    read(fd,&buf,sizeof(buf));
  }
}

void do_l1misses(int n)
{
  int i, j;

  for (j=0; j < n; j++)
    for (i=0; i < L1_MISS_BUFFER_SIZE_INTS; i++)
      buf[i] = buf[L1_MISS_BUFFER_SIZE_INTS-i] + 1;
}

void do_both(int n)
{
  int i,j;
  double c = 0.11;

  for (i=0;i<n;i++)
    {
      for (j=0; j < n; j++)
        c += a*b;
      for (j=0;j<L1_MISS_BUFFER_SIZE_INTS;j++)
        buf[j] = buf[L1_MISS_BUFFER_SIZE_INTS-j] + 1;
    }
}
```

A.2.4 zero.c

```
/* This file performs the following test: start, stop and timer functionality

   - It attempts to use the following two counters. It may use less depending on
     hardware counter resource limitations. These are counted in the default counting
     domain and default granularity, depending on the platform. Usually this is
     the user domain (PAPI_DOM_USER) and thread context (PAPI_GRN_THR).
     + PAPI_FP_INS
     + PAPI_TOT_CYC
   - Get us.
   - Start counters
   - Do flops
   - Stop and read counters
   - Get us.
*/

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include "papiStdEventDefs.h"
#include "papi.h"
#include "papi_internal.h"
#include "test_utils.h"

int main()
{
  int retval, num_tests = 1, tmp;
  int EventSet1;
  int mask1 = 0x5;
  int num_events1;
  long long **values;
  long long elapsed_us, elapsed_cyc;

  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT)
    exit(1);

  if (PAPI_set_debug(PAPI_VERB_ECONT) != PAPI_OK)
    exit(1);

  EventSet1 = add_test_events(&num_events1,&mask1);

  /* num_events1 is greater than num_events2 so don't worry. */

  values = allocate_test_space(num_tests, num_events1);

  elapsed_us = PAPI_get_real_usec();

  elapsed_cyc = PAPI_get_real_cyc();
```

```
  retval = PAPI_start(EventSet1);
  if (retval != PAPI_OK)
    exit(1);

  do_flops(NUM_FLOPS);

  retval = PAPI_stop(EventSet1, values[0]);
  if (retval != PAPI_OK)
    exit(1);

  elapsed_us = PAPI_get_real_usec() - elapsed_us;

  elapsed_cyc = PAPI_get_real_cyc() - elapsed_cyc;

  remove_test_events(&EventSet1, mask1);

  printf("Test case 0: start, stop.\n");
  printf("-----------------------------------------------\n");
  tmp = PAPI_get_opt(PAPI_GET_DEFDOM,NULL);
  printf("Default domain is: %d (%s)\n",tmp,stringify_domain(tmp));
  tmp = PAPI_get_opt(PAPI_GET_DEFGRN,NULL);
  printf("Default granularity is: %d (%s)\n",tmp,stringify_granularity(tmp));
  printf("Using %d iterations of c += a*b\n",NUM_FLOPS);
  printf("-------------------------------------------------------------------------\n");

  printf("Test type   : \t1\n");

  printf("PAPI_FP_INS : \t%lld\n",
 (values[0])[0]);
  printf("PAPI_TOT_CYC: \t%lld\n",
 (values[0])[1]);
  printf("Real usec   : \t%lld\n",
 elapsed_us);
  printf("Real cycles : \t%lld\n",
 elapsed_cyc);

  printf("-------------------------------------------------------------------------\n");

  printf("Verification:\n");

  free_test_space(values, num_tests);

  PAPI_shutdown();

  exit(0);
}
```

Bibliography

[1] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting multiple levels of parallelism in openmp: A case study. In *Proc. of the 1999 International Conference on Parallel Processing*, Aizu, Japan, sep 1999.

[2] R. Baxter. Whither hpc in europe?
a strategic review of high-performance computing from a european perspective: The direct initiative. Technical report, EPCC, `http://www.epcc.ed.ac.uk/epcc-tec/documents/tw-direct/`, August 1999.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[4] C. Brunschen and M. Brorsson. Odinmp/ccp. a portable implementation of openmp for c. *Concurrency: Practice and Experience*, pages 1193–1203, 2000. `http://www.it.kth.se/labs/cs/odinmp/ewomp99.pdf`.

[5] Rodríguez C., Sande F., León C., and García L. Parallelism and recursion in message passing libraries: An efficient methodology. *Concurrency: Practice & Experience*, 1999.

[6] Rodríguez C., Roda J.L, Sande F., Morales D. G., and Almeida F. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 2000.

[7] González D., Almeida F., Roda J., and Rodríguez C. From the theory to the tools: Parallel dynamic programming. *Concurrency: Practice & Experience*, 2000.

[8] Morales D., Almeida F., García F., Roda J.L., and Rodríguez C. Parallel dynamic programming and automata theory. *Parallel Computing*, 2000.

[9] M. Frigo. Portable high-performance programs. Technical Memo MIT/LCS/TR-785, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1999.

[10] Roda J.L., Rodríguez C., González D. G., and Almeida F. Predicting the execution time of message passing models. *Concurrency: Practice & Experience*, 1999.

[11] James Laudon and Daniel Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New YOrk, June 2–4 1997. ACM Press.

[12] OpenMP Forum. *OpenMP Architecture Review Board: OpenMP C and C++ application program interface v. 1.0*, 1998. `http://www.openmp.org/specs/ mp-documents/cspec10.ps`.

[13] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of openmp applications with nested parallelism. *Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000. `http://phase.etl.go.jp/Omni/`.

Index

(Z, L) ideal-cache model, 42

barrier, 6
bisection bandwidth, 26
bus snooping, 29

cache aware, 43
cache coherence, 29
cache coherency contention, 31
cache hit, 21
cache line, 23
cache miss, 21
cache tag field, 22
cache-coherent non-uniform memory access, 25
ccNUMA, 25
clean cache entry, 22
cray link interconnect, 26
crossbar, 24
cut-trough, 28

default, 9
diameter, 26
direct mapped cache, 23
directory-based cache coherency, 30
dirty cache entry, 22
distributed shared memory, 25
dynamic extent, 6
Dynamic Programming, 16

EventSets, 34

False sharing, 31
flits, 28
for, 9
    canonical shape, 11
fully associative, 43

general tuning problem, 42

High Performance Computing, 24
HPC, 24

hypercube, 26

independent, 14
invalidation based protocol, 30

kernel thread, 44

latency, 26
    per link, 26
Light Weight Process, 44

master, 5
master thread, 5
Memory contention, 31
minimum distance, 16

non-uniform memory access, 25
NUMA, 25

Object Oriented Programming, 15
OOP, 15
Origin 2000, 26

PAPI, 34
parallel construct, 5
parallel for, 11
parallel region, 6
Performance API, 34
private, 9
profiling, 33

race conditions, 14
re-entrant, 43
reduction problems, 19

set associative cache, 23
shared, 9
shortest path, 16
SMP, 23
spacial locality, 21
structured-block, 6
symmetric multiprocessing, 23

tall cache, 43
team, 5
temporal locality, 21
Tunable Algorithm, 42
tuning condition, 42
tuning parameters, 40

update based protocol, 29
user level threads, 44

variable-list, 9

Work Sharing Constructs, 9
wormhole routing, 28
write through, 21
write-back, 21