# Computing On Many Cores

Bernard Goossens, David Parello, Katarzyna Porada, Djallal Rahmoune

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 0000; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

# Computing On Many Cores

Bernard Goossens[1*], David Parello[1], Katarzyna Porada[1] and Djallal Rahmoune[1]

[1] *DALI, UPVD,*
*52 avenue Paul Alduy,*
*66860 Perpignan Cedex 9 France,*
*LIRMM, CNRS: UMR 5506 - UM2,*
*161 rue Ada,*
*34095 Montpellier Cedex 5 France*

SUMMARY

This paper presents a new method to parallelize programs, adapted to manycore processors. The method relies on a parallelizing hardware and a new programming style. A manycore design is presented, built from a highly simplified new core microarchitecture, with no branch predictor, no data memory and a three stage pipeline. Cores are multithreaded, run out-of-order but not speculatively and fork new threads. The new programming style is based on functions and avoids data structures. The hardware creates a concurrent thread at each function call. Loops are replaced by semantically equivalent divide and conquer functions. Instead of computing on data structures, we compute in parallel on scalars, favouring distribution and eliminating inter-thread communications. We illustrate our method on a sum reduction, a matrix multiplication and a sort. C implementations using no array are parallelized. From loop templates, a MapReduce model can be implemented and dynamically deployed by the hardware. We compare our method to *pthread* parallelization, showing that (i) our parallel execution is deterministic, (ii) thread management is cheap, (iii) parallelism is implicit and (iv) functions and loops are parallelized. Implicit parallelism makes parallel code easy to write. Deterministic parallel execution makes parallel code easy to debug.


Received ...

KEY WORDS: Computing model, Manycore processor, Parallelizing core, Deterministic parallelism, Parallel locality, Multithreading, MapReduce

## 1. INTRODUCTION

In the past decade, processors have evolved from single core CPUs to multicore and manycore processors and GPUs. Multicores started with the 2-cores Intel Core-2 introduced in july 2006 and have grown up to the 18-cores Intel Xeon E7-88x0-v3 introduced in may 2015. Nvidia NV1 was the first GPU, with only one core, launched in september 1995. The Nvidia GM200 launched in march 2015 has 3072 cores. Manycores were introduced by the Tilera Tile64 (64-cores) in 2007. Intel followed with the 61-cores Xeon Phi (introduced with a 32-cores in 2010). Kalray proposed in 2014 the 256-cores MPPA which was recently upgraded with the 288-cores MPPA2.

These different industrial products reflect three conceptions of parallel programming. The multicore processors (from 2-cores to a few tens) address general purpose sequential applications. The manycore processors (from a few tens to a few hundreds) concern parallel applications. The GPUs (a few thousands) are used for vectorized computations.

*Correspondence to: Email: goossens@univ-perp.fr

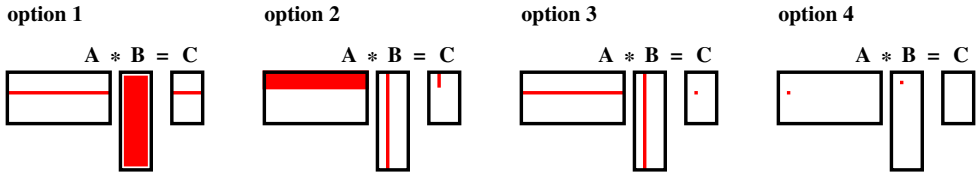


Figure 1. Four threads organizations to compute a matrix multiplication C=A*B in parallel

For a given problem, how can we match its programmed solution to the best suited processor? Should we write a sequential program run on a single thread of a multicore processor? Would it be cost-efficient to parallelize the code and run it on a manycore processor? Or can we find enough regularities in the data to vectorize the solution and implement it on a GPU? A fourth option could be to decompose the problem and have multiple pieces of code running on each of the three types of parallel processors, using an heterogeneous computer [1].

Each of the four possibilities leads to use different programming languages (Java, High Parallel Fortran, Glasgow Parallel Haskell, Cilk, Cuda, Open-CL), libraries (Pthreads, Open-MP, MPI) and combining tools (MPI+OpenMP, Cuda+OpenMP, Cuda+MPI). The final result, i.e. the program solving the initial problem, is likely to be tightly binded to the target computer. Migrating from a multicore processor to a manycore one, a GPU or a hybrid core computer means rewriting most of the code. Adapting programs to new processors is not done through a simple recompilation.

Is this situation inherent to the variety of problems or does it come from systems which are not abstract enough to be applicable to the whole range of computational problems?

In this paper, we advocate for reconsidering both the hardware and the software, to provide a single frame into which parallel programs can be quickly designed and safely run.

To illustrate some aspects of the actual parallel programming choices complexity, we consider the matrix multiplication problem. Figure 1 shows, among a full range, four organizations of the threads computing a matrix multiplication C[m,p]=A[m,n]*B[n,p], in increasing parallelism order. In each subfigure, the red part represents the sources and results of a single thread. Option one uses one thread per matrix C line. Option two has one thread for a subset of one matrix C column. Option three has one thread per matrix C element. Option four has one thread per multiplication.

These organizations result in four quite different programs, would they be coded with the *pthread* library, MPI or openMP (GPUs are not further considered in the paper). This is the consequence of the explicit parallelism, i.e. thread creations, synchronizations and communications.

In each version there are different programming difficulties. In the fourth option a result matrix element must be shared by the $n$ threads writing to it. In the second option, the program should pay attention to the false sharing at the junction of two consecutive subsets of a result column, i.e. the size of each thread computation should be adapted to the cache organizations.

There is no best choice. It depends on the number of cores, the thread creation and synchronization cost, the communication cost which itself depends on the cores and memory topology. Even worse, a choice can be good today and bad later, with hardware and OS upgrades.

The rightmost computation seems to be the least efficient because it involves too small threads and the writers must be synchronized. But this organization captures all the available data parallelism.

In this paper, we describe a new parallel programming model aiming to replace actual thread based models. The proposed model relies on a parallelizing hardware briefly presented in section 2 and on a new programming style developed in section 3. The proposed hardware is based on a simplified out-of-order core. Our processor is made of many small cores using registers rather than memory, as a GPU, but it is MIMD rather than SIMD.

The programming model relies on a standard programming language like C with a gcc-like compiler suite. The parallelization, instead of being static (compiler parallelizing directives) or semi-static (compiler directives + OS primitives), is fully dynamic, i.e. parallelism is implicit in a code written with a standard sequential language. The syntactical order of the instructions in the code fixes the deterministic order of the computation, i.e. the sequential semantic. The program is



DOI: 10.1002/cpe

run in parallel thanks to the parallelizing hardware. The parallel run is reproducible because the parallel semantic is equivalent to the sequential semantic.

A program written in a high level language like C can be interpreted as a parallel program when the function call instruction semantic is slightly changed, assuming it forks. A resume thread is created in parallel with the calling one. The core hardware implements the forking semantic of the call instruction. Each core is multithreaded (like Intel Hyperthreading [2]) and communicates with its neighbours through a bidirectional ring. At function call, a free thread slot is allocated on the successor core to host the resume thread which is fetched in parallel with the main thread.

Parallelism is implicitly deployed and managed by the hardware. Communications and synchronisations are implicitly derived from producers to consumers dependences. A reader is matched with and synchronized to its unique writer by hardware renaming [3].

The model has many advantages among which:

- a run is parallel and deterministic,
- parallelism is implicit,
- loops are parallelized,
- no OS overhead,
- no hypothesis on the number of available resources,
- easy debugging of the parallelized code.

Hardware parallelization is compared to OS parallelization in section 4. Section 5 places our proposition in the context of actual parallelizing methods and tools and concludes.

## 2. A PARALLELIZING HARDWARE

Figure 2 shows the manycore processor design. The left part of the figure is the general topology of a 32-core processor and the right part is the inside of a core. The cores are linked by a bidirectional ring (magenta color). A core communicates with its successor and predecessor: the send unit in green is linked to the predecessor and to the successor receive units in red. The processor has a set of shared L2 caches which hold code and I/O data (the L2 access buses are in cyan color).

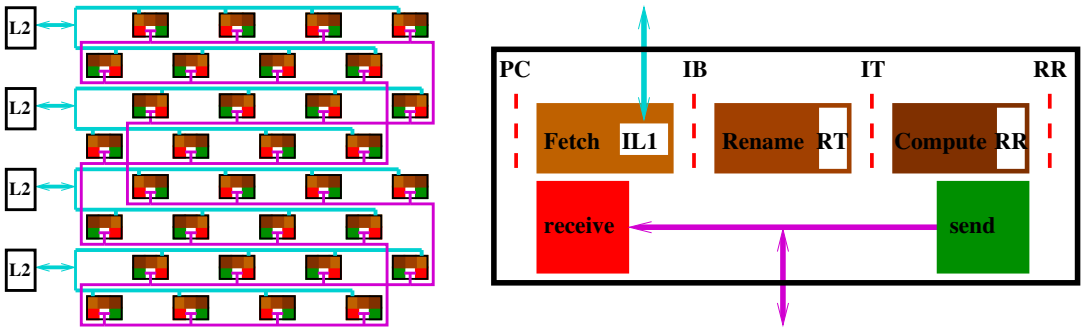


Figure 2. A manycore processor

Each core hosts a set of threads (e.g. 16 threads per core). A thread is represented by its PC and its renaming table (RT). The core pipeline has three stages (right part of the figure). The fetch stage selects a ready thread PC to fetch one instruction from IL1. The instruction is saved in the thread instruction buffer (IB). The rename stage selects a full IB and the instruction it holds is decoded and renamed through the thread RT. The renamed instruction is saved in the instruction table (IT). The compute stage selects one ready instruction in IT which is executed: it reads its sources from and writes its result to the Renaming Registers (RR). At full speed, the processor runs one instruction per cycle (IPC) per core (e.g. 1K IPC for a 1K-core processor).

The Instruction Set Architecture (ISA) is restricted to register-register, control and I/O instructions. There are no memory-access instructions.



DOI: 10.1002/cpe

```
int sum(int i, int n){
 if (n==1) return f(i);
 if (n==2) return f(i)+f(i+1);
 return sum(i,n/2) +
        sum(i+n/2,n-n/2);
}
```

```
void main(){
 printf("s=%d\n",sum(0,10));
}
inline int f(int i){
 return i;
}
```

Figure 3. A vector sum reduction programmed in C

```
1  sum:  cmpq   $2, %rsi          /*  if (n>2)                  */
2        ja     .L2              /*    goto .L2                */
3        movq   %rdi, %rax       /*  rax  = f(i)               */
4        subq   $1, %rsi         /*  if (n==1)                 */
5        je     .L1              /*    goto .L1                */
6        addq   $1, %rdi         /*  rdi  = f(i+1)             */
7        addq   %rdi, %rax       /*  rax  = f(i)+f(i+1)        */
8  .L1:  ret                     /*  stop                      */
9  .L2:  movq   %rsi, %rbx       /*  rbx  = n                  */
10       shrq   %rsi             /*  rsi  = n/2                */
11       fork   $3               /*  start thread              */
12       push   %rdi             /*  send rdi                  */
13       push   %rsi             /*  send rsi                  */
14       push   %rbx             /*  send rbx                  */
15       call   sum              /*  rax  = sum(i,n/2)         */
16       pop    %rbx             /*  receive rbx               */
17       pop    %rsi             /*  receive rsi               */
18       pop    %rdi             /*  receive rdi               */
19       movq   %rax, %rcx       /*  rcx  = rax                */
20       addq   %rsi, %rdi       /*  rdi  = i + n/2            */
21       subq   %rsi, %rbx       /*  rbx  = n - n/2            */
22       movq   %rbx, %rsi       /*  n    = n - n/2            */
23       fork   $1               /*  start thread              */
24       push   %rcx             /*  send rcx                  */
25       call   sum              /*  rax  = sum(i+n/2,n-n/2)   */
26       pop    %rcx             /*  receive rcx               */
27       addq   %rcx, %rax       /*  rax += sum(i,n/2)         */
28       ret                     /*  stop                      */
```

Figure 4. A vector sum reduction translated into *x86*

Each core has two special units to send and receive messages to and from its neighbours. A message is sent to the prior or next thread, hosted by a neighbour core (mostly, the successor). A message contains a register. A new thread Program Counter (PC) value is sent to the successor core when a call instruction is decoded. A register source $r$ which is not locally set is imported from the core hosting the prior thread.

Figure 3 is a vector sum reduction programmed in C and figure 4 is its translation in x86. Function $f(i)$ returns vector element $i$. The code does not implement the vector as an array but as a function returning any of its elements (function $f$ could, instead of providing the value itself, get it from an input file; the OS I/O driver should allow parallel I/O, like in MPI2 [4]). The x86 translation does not use any memory access. The computation is done within the set of architectural registers.

The *fork $k* instruction creates a new thread on the successor core, starting from the resume address after the next *call* instruction. In between are $k$ *push r* instructions. The *push r* instruction sends a copy of register $r$ to the new thread. The creating thread sends $k$ values to the created thread.

For example, the *fork $3* on line 11 starts a remote thread. The next three *push* instructions send copies of registers $rdi$, $rsi$ and $rbx$ to the created thread. The *call* instruction sends the resume code address, i.e. a copy of the return PC. Once the core hosting the new thread has received the



DOI: 10.1002/cpe

3 registers and the PC value, it starts fetching. The thread running the *call* on line 15 jumps to the label target, i.e. to line 1 and the created thread fetches from line 16 in parallel.

The compiler inserts a *pop r* instruction to receive register $r$ in the resume thread. For example, the resume thread started at line 16 receives registers $rbx$, $rsi$ and $rdi$ sent by the main thread.

The send/receive machine instructions names are *push/pop* which can seem confusing. There is no stack involved if the run is parallelized, only a value transmission (sent by *push* and received by *pop*). A $push$ instruction is turned into a $send$ operation, which waits for the pushed register value and then sends it to the destination. A $pop$ instruction is turned into a $receive$ operation, which waits for the transmitted value and then writes it to the register destination. Push and pop instructions may be run out-of-order. The reception order is irrelevant.

The presence of the *push/pop* instructions allows the hardware to switch between parallel and sequential modes. The *fork* instruction, which creates a remote thread, is blocking until a free slot has been allocated. If the instruction is run by the oldest thread and if no slot is free in the selected core, a fail message is immediately sent back and the creating thread switches to sequential mode. The *call*, *ret* and *push/pop* instructions regain their original stack related semantic. Each thread slot has a fixed size private stack, which expands in L2. The thread switches back to parallel mode when a *ret* instruction empties the stack. The oldest thread special behaviour guarantees that it may not be blocked forever, ensuring a deadlock free parallel execution.

When run in parallel mode, a *ret* instruction stops its thread. In parallel mode, the *call* and *ret* instructions do not save/restore the return address on the stack.

Figure 5 shows the parallelization of 11 threads summing a 10 integer vector. Each thread is surrounded by a red rectangle. For example, thread 1 runs 26 instructions (9+9+8) on core 0. The *ret* instruction on line 8 ends the thread.

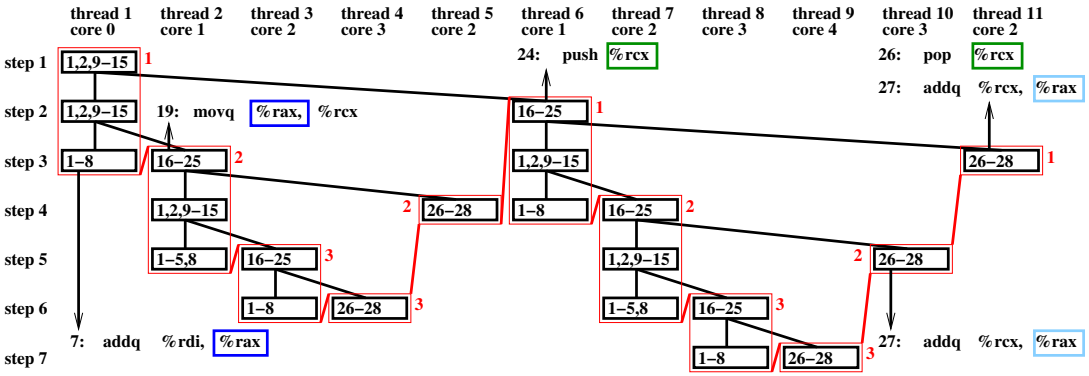


Figure 5. Trace of *sum(0, 10)*

When the *fork $3* instruction on line 11 is run, a new thread is started (thread 6, on successor core 1). When the *call* instruction on line 15 is run, the return address (line 16) is sent to the created thread as its starting fetch address. Thread 1 starts successively threads 6 and 2, both at line 16.

Every thread is linked by the hardware to its predecessor and successor (red lines on the figure). When the successor links are followed, the sequential trace is built. Threads have a hierarchical level, depicted in red figures on the top right corner of the threads surrounding rectangles. The higher the level, the lower its figure. For example, threads 1, 6 and 11 form highest level 1.

The eleven threads are deployed in seven successive steps in the manycore thread slots.

As soon as a thread has executed all its instructions, it frees its hosting slot. In the example given on figure 5, threads 3 and 8 are first freed, then 4 and 9, eliminating level 3. The oldest thread (i.e. thread 1) is freed in parallel with 3 and 8. A thread can be freed if it is the oldest or if its predecessor has a higher level and its successor has a higher or equal level.

During the run the tree of threads expands on the right and the bottom (new threads) and contracts on the left and the bottom (ended threads). If a core gets saturated with threads, the expansion stops there while no free slot is available but the contraction continues, freeing slots.



DOI: 10.1002/cpe

Concerning synchronizations and communications, the hardware matches a reader with its writer through register renaming. The written value is copied to the reading source. For example on figures 4 and 5, instruction 7 in thread 1 writes the sum f(0)+f(1) into register $rax$ (lower blue box on figure 5). Instruction 19 $rax$ source in thread 2 (upper blue box) is matched with instruction 7 $rax$ destination in thread 1, as it is the first $rax$ writer met on the backward travel along the ordered threads, starting from thread 2. In the same manner, instruction 27 $rax$ source in thread 11 (upper light blue box) is matched with instruction 27 $rax$ destination in thread 10 (lower light blue box). In a different way, instruction 26 $rcx$ popped in thread 11 (rightmost green box) is matched with instruction 24 $rcx$ pushed in thread 6 (leftmost green box). In this case, the $rcx$ value is directly sent from thread 6 (core 1) to thread 11 (core 2) when computed. All these communications concern neighbour cores.

## 3. A NEW PROGRAMMING STYLE

To take advantage of the parallelizing hardware, the programmer should adopt a new programming style, including the following requirements:

- decompose the code into functions,
- translate *for* and *while* loops into divide-and-conquer template functions,
- avoid separating inputs and outputs from computations,
- recompute rather than store and load,
- avoid data structures, only use scalars: no memory, no pointer, no array, no structure.

This is illustrated by a parallelized matrix multiplication and a parallelized sort.

The programming style we describe looks like the functional programming paradigm [5] [6] [7]. The program examples are written in C and most of them would be more elegant in Haskell. We have choosen C and x86 to keep close to the hardware.

The programming style we use is in some way more restrictive than the functional paradigm. A program is a composition of functions which only compute scalars, rather than lists like in Lisp. We want the computation to be optimally distributed, which requires to compute on scalars instead of data structures. The programming style we propose is close to lambda-calculus [8], with expansion, i.e. substitution, and reduction. But it may allow a restricted form of assignment, to save to registers and later reuse intermediate computed scalars.

The general programming pattern is to organize a parallel computation from each result to output and backward up to the inputs it uses. Each result is produced by an independent computation, using its own copies of the inputs. Each computation is a succession of transformations from the used inputs to the target output, with no intermediary structured storing.

If a set of scalar results are to be used more than once, each scalar may either be individually recomputed or register saved and later restored. But the structured set of data should not be built in memory and loaded because storing a structure in a manycore centralizes its data. In a manycore, it is slower to gather a data structure, keep it in memory and later scatter its components than to compute and use elements separately. It is also probably less energetically efficient because of the energetical cost of memorization and communication. If the program should compute on structured data, the data should only be structured externally from the computation. A data structure is input in parallel and its scalar components are distributed to the set of parallel read instructions. The scalar elements composing an output data structure are written in parallel. The parallel computation itself does not manipulate any data structure.

The hardware is composed of some external memory holding the code to be run and the structured input and output data. This external memory has a high throughput to allow for parallel I/O requests from the processor. The processor is the one described in section 2, using private and shared caches to shorten the external memory access latency. Input machine instructions move the data from the external input to the processor, where the results are computed, which are sent back to the external memory with output machine instructions. The memory hierarchy is naturally coherent as output destinations are written only once.



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

The sum reduction program given on figure 3 is a composition of scalar sums. There is no vector in the computation. The scalars to be summed are input when needed, i.e. when the $sum$ function reaches a recursion stop condition to sum up two adjacent values. They are given by an initializing function $f$ returning the vector value for index $i$. The final sum is sent to output.

The sum reduction does not use any storing memory. Intermediate sums are computed in allocated renamed registers, which are freed with their computing thread. Each core contains 128 renaming registers (RR file on figure 2, right), shared by its 16 thread slots. A 1K core processor has 128K registers, making data memory unnecessary to hold intermediate results.

As the programming style avoids memory storing and loading, communications are reduced to the sent/received registers and the result transmission from a main thread to a resume one. Because a complex computation is decomposed into scalar computations, the run is fully distributed. The thread creation being done by the hardware, there is no overhead and parallelization can be fine grain. Eventually, thread ordering and renaming ensures a deterministic computation because the partial order run preserves the producer to consumer dependences of the sequential order.

A consequence of the implicit parallelism is that the parallel code can be tested on a sequential machine. The deterministic execution makes debugging as easy as for a sequential run. Bugs are reproducible. As an illustration, all the C codes in this paper can be compiled with a standard $gcc$ compiler and their run on a sequential processor produce the same results in the same order as the parallel run would on a parallelizing hardware, thanks to determinism.

```
//for (i=lower; i<upper;
//      i++) body(i,arg);
//to parallelize the loop
//replace it by a call to
//for_loop(lower,
//    upper-lower,body,arg);
```

```
void for_loop(int i, int n, void (*body)(),
 void *arg){
 if (n==1){body(i, arg); return;}
 if (n==2){body(i, arg); body(i+1, arg); return;}
 for_loop(i, n/2, body, arg);
 for_loop(i+n/2, n-n/2, body, arg);
}
```

Figure 6. *For* loop template function

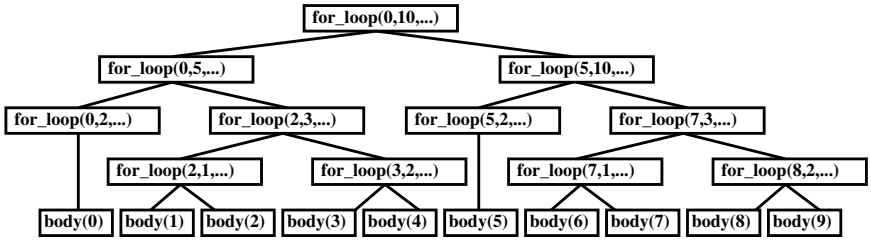


Figure 7. Threads created by a 10 iterations parallelized *for* loop

The time complexity of algorithms and programs in the proposed parallel execution model is not measured in terms of operations but in terms of depending threads. For example, the sum reduction complexity is O($log\ n$), which means that $log\ n$ threads launching steps are needed to deploy the code on a hardware having enough available thread slots.

### *3.1. Parallelizing loops*

To be parallelized by our hardware, a loop must be written as a function.

*3.1.1.* For *loops.* Figure 6 shows a *for* loop template function. Each loop parallelized this way has a O($log\ n$) complexity to deploy the threads running the $n$ iterations. An iteration excluding condition is added as an argument on figure 8 right template.

Figure 7 shows the threads created by a 10 iterations parallelized *for* loop.



DOI: 10.1002/cpe

The figure 6 template function uses pointer $arg$ to encapsulate the $body$ function arguments. This way to transmit an unknown number of values was choosen to keep the code close to the $pthread$ usage. As the parallelizing hardware we propose does not have any memory access instruction in the ISA, pointers and structures are not available. The encapsulated arguments should be interpreted as a list of scalar values rather than a C structure, as a $va\_list$ type provided by $stdarg.h$.

```
//i=lower;
//while (!cond(i,arg_cond)){
// body(i,arg_body); i++;
//}
//to parallelize the loop
//replace it by a call to
//while_loop(lower,1,
// cond,arg_cond,body,arg_body);

//returns the number of
//iterations in the while
//launches n, 2*n, 4*n ...
//iterations until cond
int while_loop(int i, int n,
 int (*cond)(), void *arg_cond,
 void (*body)(), void *arg_body){
 int nb_iter=
   for_cond(i,n,
   cond,arg_cond,body,arg_body);
 if (nb_iter==n)
   nb_iter+=
   while_loop(i+n,2*n,
   cond,arg_cond,body,arg_body);
 return nb_iter;
}
```

```
//launches n iterations
//returns the number of iterations
//not excluded by the cond function
int for_cond(int i, int n,
          int (*cond)(), void *arg_cond,
          void (*body)(), void *arg_body){
 int c1, c2;
 if (n==1){
  c1=cond(i,arg_cond);
  if (!c1) body(i,arg_body);
  return !c1;
 }
 if (n==2){
  c1=cond(i,arg_cond);
  if (!c1) body(i,arg_body);
  c2=cond(i+1,arg_cond);
  if (!c2) body(i+1,arg_body);
  return (!c1) + (!c2);
 }
 c1=for_cond(i,n/2,cond,arg_cond,
              body,arg_body);
 c2=for_cond(i+n/2,n-n/2,
              cond,arg_cond,body,arg_body);
 return c1+c2;
}
```

Figure 8. *While* loop and conditional *for* loop template functions

*3.1.2.* While *loops.* The left part of figure 8 shows the *while* loop template function. The $for\_cond$ function shown on the right part of the figure is a *for* loop with an exclusion condition $cond$. It returns the number of non excluded iterations. The *while* loop runs $n$ iterations in a $for\_cond$ loop. If no iteration is excluded by the $for\_cond$, the $while\_loop$ function is called recursively to run $2 * n$ more iterations. It runs 1 iteration, then 2, 4, 8 ... until the $for\_cond$ reaches the $cond$ condition. It returns the number of iterations in the *while* loop. Each *while* loop parallelized this way also has a O($log\ n$) complexity to deploy the threads running the $n$ iterations.

Figure 9 shows two applications, the left one using figure 8 *while_loop* pattern. On the left of the figure is the parallelized computation of $my\_strlen$. The $while\_loop$ function runs 1+2+4+8 iterations. The $for\_cond$ called for 8 iterations returns 4, which stops the $while\_loop$ recursion. The computed string length is 1+2+4+4=11. (It may seem tedious to write a function like $s(i)$ for each declaration of an initialized array. The compiler can be adapted to translate such declarations into functions). On the right of figure 9 is a parallelized computation of a search of a character in a string. The $my\_strchr$ function uses the $for\_reduce$ pattern defined on figure 11.

The parallelization of a *while* loop launches iterations after the exit condition. In the *while* loop pattern, it is assumed that when $cond$ is true, it remains true for the following iterations, which are all excluded by the $for\_cond$ loop. It works for the $my\_strlen$ function example (left of figure 9) because accessing beyond the string end returns $'\backslash 0'$. It does not work for the $my\_strchr$ function (right part of the figure). In a search, iterations after the searched element are parasitic and should be explicitely excluded. The loop is parallelized by the $for\_reduce$ function, which is a *for* loop computing a reduction. The length $l$ of the searched string $s$ is computed and the searched character is compared in parallel to each character of $s$. The reduction returns the leftmost match index. If the searched character is not found, the reduction returns $l + 1$.



DOI: 10.1002/cpe

```
//to compute strlen(char *s)
//char s[]="hello world";
//the compiler builds function s(i)
typedef struct {int (*s)();} ArgC;
int s(int i){
 switch(i){
  case 0: return 'h'; case 1: return 'e';
  case 2: return 'l'; case 3: return 'l';
  case 4: return 'o'; case 5: return ' ';
  case 6: return 'w'; case 7: return 'o';
  case 8: return 'r'; case 9: return 'l';
  case 10: return 'd'; default: return '\0';
 }
}
int exit_cond(int i, void *arg_cond){
 ArgC *a=(ArgC *)arg_cond;
 return ((a->s)(i)=='\0');
}
void null_body(int i, void *arg_body){}
int my_strlen(int (*s)()){
 ArgC ac; ac.s=s;
 return while_loop(0,1,exit_cond,(void*)&ac,
                   null_body,NULL);
}
void main(){printf("%d\n",my_strlen(s));}
```

```
//my_strchr(s,c) is the position
//of first c in s
typedef struct {char c; int l;
                int (*s)();} Arg;
int min(int i, int n,
     int a, int b, void *arg){
 if (a<b) return a;
 else return b;
}
int found(int i, void *arg){
 Arg *a=(Arg *)arg;
 if (a->s(i)!=a->c) return a->l;
 else return i;
}
int my_strchr(int (*s)(),
 char c){
 Arg a; int l=my_strlen(s);
 a.c=c; a.l=l; a.s=s;
 return for_reduce(0, l, l+1,
  found,(void *)&a,min,NULL);
}
main(){
 printf("first o at %d\n",
        my_strchr(s,'o'));
}
```

Figure 9. Parallelizing $my\_strlen$ (left) and $my\_strchr$ (right)

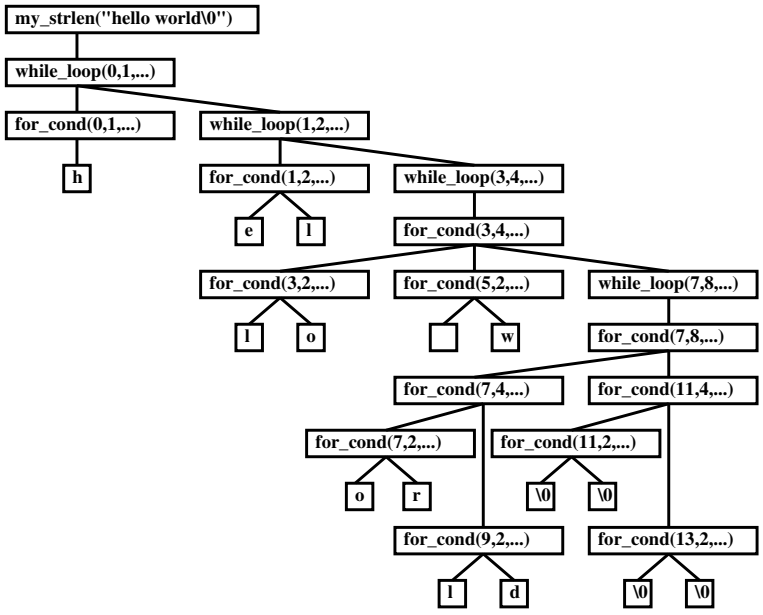


Figure 10. Threads created by the run of $my\_strlen(\text{"}hello\ world\backslash 0\text{"})$

As the *for_loop*, the *while_loop*, the *for_cond* and the *for_reduce* template functions use pointer arguments to be interpreted as lists of scalars, transmitted from the caller to the callee through registers.

Figure 10 shows the threads created by the $my\_strlen(\text{"}hello\ world\text{"})$ execution.



DOI: 10.1002/cpe

*3.1.3. Reduce* for *loops.* Figure 11 shows the template of a reduction *for* loop and a revisited version of the vector sum reduction. Any similar template can be written to provide a *while_reduce* and all the necessary tools to build a MapReduce program [9].

The $get$ and $reduce$ functions in $for\_reduce$ and $cond$ and $body$ functions in $for\_loop$, $for\_cond$ and $while\_loop$ should be carefully programmed to avoid any serialization of the run, as we have done in the given examples. It would be a design error to increment a counter in the $my\_body$ function of the $my\_strlen$ example to compute the length. The hardware would run the loop correctly but serially because of the recurrence in the iteration body.

```
//rnv is the reduction neutral value
int for_reduce(int i, int n, int rnv,
 int (*get)(), void *arg_get,
 int (*reduce)(), void *arg_reduce){
 if (n==1) return reduce(i, n,
     get(i,arg_get), rnv, arg_reduce);
 if (n==2){
  return reduce(i, n, get(i,arg_get),
         get(i+1,arg_get), arg_reduce);
 }
 return reduce(i, n,
  for_reduce(i, n/2, rnv, get, arg_get,
         reduce, arg_reduce),
  for_reduce(i+n/2, n-n/2, rnv, get,
         arg_get, reduce, arg_reduce),
  arg_reduce);
}
```

```
typedef struct {int (*f)();} Arg;
int v(int i){return i;}
int sum(int i, int n, int a, int b,
        void *arg){return a+b;}
int get(int i, void *arg){
 Arg *a=(Arg *)arg; return a->f(i);
}
int sum_reduction(int (*get)(),
    void *arg_get, int n){
 return for_reduce(0,n,0,
      get,arg_get,sum,NULL);
}
main(){
 Arg a; a.f=v;
 printf("sum=%d\n",sum_reduction(
          get,(void *)&a,SIZE));
}
```

Figure 11. A reduce *for* loop template and a revisited version of *sum*

```
int a[2][3]={{1,2,3},{0,1,2}},
    b[3][4]={{2,3,4,5},{3,2,1,0},
             {0,1,2,3}},
    c[2][4];
void print_mat(int *a, int m, int n){
 int i,j;
 for (i=0;i<m;i++){
  for (j=0;j<n;j++)
   printf("%d ",*(a+i*n+j));
  printf("\n");
 }
}
void imatmul(int m, int n, int p){
 //c[m][n] = a[m][p] * b[p][n]
 int i, j;
 for (i=0; i<m; i++)
  for (j=0; j<n; j++)
   *((int*)c+i*n+j) = sum(0,p,i,j,n,p);
}
```

```
int sum(int fk, int nk, int i,
        int j, int n, int p){
 if (nk==1)
  return *((int*)a+i*p+fk) *
         *((int*)b+fk*n+j);
 if (nk==2){
  return
   (*((int*)a+i*p+fk) *
    *((int*)b+fk*n+j)) +
   (*((int*)a+i*p+fk+1) *
    *((int*)b+(fk+1)*n+j));
 }
 return sum(fk,nk/2,i,j,n,p) +
        sum(fk+nk/2,nk-nk/2,i,j,n,p);
}
main(){
 print_mat((int*)a,2,3);
 print_mat((int*)b,3,4);
 imatmul(2,4,3);
 print_mat((int*)c,2,4);
}
```

Figure 12. A matrix multiplication programmed in C

### *3.2. A Parallelized Matrix Multiplication*

*3.2.1. The classical matrix multiplication program.* Figure 12 shows the C code of a matrix multiplication. The classical matrix multiplication algorithm is a good illustration of how the parallelizing hardware can parallelize nested *for* loops.



DOI: 10.1002/cpe

If we use the program on figure 12 as a canvas for a parallelized implementation, the parallel program will probably be organized in three different phases. The first phase sets the inputs, the second phase computes the product and the third phase outputs it. The first phase appears in the sequential code as matrix $a$ and $b$ initializations. This job is done by the OS at process start, by copying the data segment from the ELF file into the memory. The third phase sends each element of the product matrix to the OS output buffer (logical output driver called by $printf$) and from there, the buffer content is sent to the physical driver (either a display or a file).

If the OS is not parallelized (e.g. Unix), I/O are sequential, which sequentializes phases one and three. Otherwise, the three phases may all be parallelized.

To avoid serializations, input and computation may be fused (i.e. start some computation product as soon as their input data are set) as well as computation and output (i.e. output one element as soon as it is computed). Input matrix elements are copied from file to memory in parallel. In parallel with the inputs, products are computed with the available data from memory. In parallel with the products computations, the computed sums of products are copied to the output file.

Even though we parallelize this much, we have a big communication problem. The input matrices are centralized in the core which runs the loader $\_start$ function. Each element is consumed by many products, which can be distributed on many cores, requiring many communications from the owner cache to the consumers ones. The same communication problem applies between the product producers and the vector sum consumers, with the aggravating difficulty of coherent cache updates.

The parallelizing processor we have described in section 2 runs the four function calls in $main$ in parallel (figure 12, bottom right part). It parallelizes the input matrix printing, the product matrix computation and its printing. However, it does not parallelize the *for* loops as iterations are not functions. The classical matrix multiplication program is not suited to our parallelizing hardware.

*3.2.2. Parallelizing the matrix printing and the vector product.* Figures 13 and 14 show functions $get\_a$ and $get\_b$ to read one element of matrices $a$ and $b$. Instead of reading the input values from the data memory, the threads read them from the code memory (or from a file), which can be duplicated and cached in all the requesting cores.

```
int get_a(int i, int j){
 switch(j){
  case 0: if (i==0) return 1; else return 0;
  case 1: if (i==0) return 2; else return 1;
  case 2: if (i==0) return 3; else return 2;
 }
}
```

```
typedef struct {int fj; int nj;
  int n; int p; void (*body_j)();
  int (*get_m)();} Arg_i;
typedef struct {int i; int n;
  int p; int (*get_m)();} Arg_j;
```

Figure 13. Function to get matrix $a$ elements

```
int get_b(int i, int j){
 switch(i){
  case 0: switch(j){
   case 0: return 2; case 1: return 3;
   case 2: return 4; case 3: return 5;}
  case 1: switch(j){
   case 0: return 3; case 1: return 2;
   case 2: return 1; case 3: return 0;}
  case 2: switch(j){
   case 0: return 0; case 1: return 1;
   case 2: return 2; case 3: return 3;}
 }
}
```

```
void body_pr_j(int j, void *arg){
 Arg_j *a=(Arg_j *)arg;
 printf("%d ",a->get_m(a->i,j));
 if (j==a->p-1) printf("\n");
}
void body_pr_i(int i, void *arg){
 Arg_i *ai=(Arg_i *)arg; Arg_j aj;
 aj.i=i; aj.n=ai->n;
 aj.p=ai->p; aj.get_m=ai->get_m;
 for_loop(ai->fj,ai->nj,
          ai->body_j,(void *)&aj);
}
```

Figure 14. Functions to get matrix $b$ elements and to print a matrix



DOI: 10.1002/cpe

The right part of figure 14 shows the two nested loops to print a matrix. It illustrates the way the $for\_loop$ template function implements nested loops.

*3.2.3. Parallelizing the matrix multiplication.* On figure 15 in the $body\_mm$ function, the computed sum $s$ should be stored into an array element saving $C[i,j]$. Element $C[i,j]$ is consumed by the printing function rather than stored. The threads sequential ordering ensures that even though the $C$ values are computed out-of-order, they are output in order. It is easy to check that when the code is sequentially run, matrix $C$ is properly printed. As the parallel run preserves sequential dependences, the computed $C$ values can be written in order by an *ad hoc* OS output driver in the video memory.

```
void body_mm(int i, int j, int n,
             int p){
 ArgProd a;
 int s; a.i=i; a.j=j;
 s=sum_reduction(prod,(void *)&a,p);
 printf("%d ",s);
 if (j==n-1) printf("\n");
}
void my_body_mm_j(int j, void *arg){
 Arg_j *a=(Arg_j *)arg;
 body_mm(a->i,j,a->n,a->p);
}
void my_body_mm_i(int i, void *arg){
 Arg_i *ai=(Arg_i *)arg;
 Arg_j aj;
 aj.i=i; aj.n=ai->n; aj.p=ai->p;
 for_loop(ai->fj,ai->nj,ai->body_j,
          (void *)&aj);
}
```

```
typedef struct {int i; int j;} ArgProd;
int prod(int i, void *arg_prod){
 ArgProd *a=(ArgProd *)arg_prod;
 int le, ri;
 le=get_a(a->i,i);
 ri=get_b(i,a->j);
 return le*ri;
}
main(){
 Arg_i a;
 a.fj=0; a.nj=3; a.n=2; a.p=3;
 a.body_j=body_pr_j; a.get_m=get_a;
 for_loop(0,2,body_pr_i,(void*)&a);
 a.fj=0; a.nj=4; a.n=3; a.p=4;
 a.body_j=body_pr_j; a.get_m=get_b;
 for_loop(0,3,body_pr_i,(void*)&a);
 a.fj=0; a.nj=4; a.n=4; a.p=3;
 a.body_j=body_mm_j;
 for_loop(0,2,body_mm_i,(void*)&a);
}
```

Figure 15. Computing the matrix product and $main$ function

The run is fully distributed to capture all the available data parallelism.

```
#define SIZE 10
struct node{
  int i;
  struct node *l; struct node *r;
};
insert(int i, struct node **r){
 if (*r == NULL){
  *r = (struct node*)
       malloc(sizeof(struct node));
  (*r)->i = i;
  (*r)->l = NULL; (*r)->r = NULL;
 }
 else if (i < (*r)->i){
  insert(i, &(*r)->l);
 }
 else insert(i, &(*r)->r);
}
```

```
int v[SIZE]={3,0,1,5,9,7,6,3,4,1};
struct node *r = NULL;
void travel(struct node *r){
  if (r != NULL){
   travel(r->l);
   printf("%d ", r->i);
   travel(r->r);
   free(r);
  }
}
main(){
 int i;
 for (i=0; i<SIZE; i++)
  insert(v[i], &r);
 travel(r);
 printf("\n");
}
```

Figure 16. A binary tree sort programmed in C

To summarize, the parallelized matrix multiplication $C[m,p] = A[m,n] * B[n,p]$ is a set of $m*n*p$ threads, each computing a single product. $C[i,j]$ is a sum reduction of the $k$ products $A[i,k] * B[k,j]$. The input matrices are not organized as arrays. Their elements are given by two access functions $get\_a$ and $get\_b$ returning $A[i,j]$ and $B[i,j]$ for indexes $i$ and $j$. The resulting matrix



DOI: 10.1002/cpe

is neither stored as an array. Each element is sent to output as soon as computed. The complexity is O($(log\ m) * (log\ n) * (log\ p)$), requiring $(log\ m) * (log\ p)$ steps to deploy all the threads to compute the $m * p$ elements of matrix C and from there, $log\ n$ steps to sum $n$ products by a reduction.

### *3.3. A Parallelized Sort*

Figure 16 shows a binary tree sort programmed in C. This is not the best sorting algorithm to be parallelized but it emphasizes the potential communications problems. The process running the program sets the initial unsorted vector in the data region (OS copy from the ELF file). The *for* loop inserts each element of the vector into a binary tree. The *travel* function copies each element of the tree into the output buffer.

Concurrent updates of a tree require complex mutual exclusion, especially if we want to interleave the tree construction, its travelling and its destruction to fully parallelize the sort. Even if a satisfying parallel code is built, its run requires three copies of each element (from ELF file to data region, then to tree and from tree to output buffer). These copies involve communications which might be long distance in a manycore processor. Binary tree writes imply complex memory coherence control.

Figure 17 shows the $main$ and $print\_body$ functions of a parallel sort with no array. The input data are given by function $my\_random$, delivering a precomputed random value. The random value precomputation avoids the pseudo-random suite dependences. The $main$ function prints the initial set of unsorted values and in parallel prints the sorted set.

```
#define SIZE 10
int my_random(int i){
 switch(i){
  case 0: return 3; case 1: return 0;
  case 2: return 1; case 3: return 5;
  case 4: return 9; case 5: return 7;
  case 6: return 6; case 7: return 3;
  case 8: return 4; case 9: return 1;
 }
}
```

```
typedef struct {int i; int v;
                int (*f)();} Arg;
void print_body(int i, void *arg){
 printf("%d ",my_random(i));
}
void main(){
 for_loop(0,SIZE,print_body,NULL);
 printf("\n");
 for_loop(0,SIZE,sort_body,NULL);
 printf("\n");
}
```

Figure 17. A parallel sort programmed in C: $main$, initial element print and $my\_random$ input functions

```
int get(int i, void *arg){
 Arg *a=(Arg *)arg;
 return (a->f(i)<a->f(a->v));
}
int get_next(int i, void *arg){
 Arg *a=(Arg *)arg;
 return (a->f(i)<=a->f(a->v));
}
int for_pos(int i, int n, int v,
 int (*f)()){
 Arg a; a.f=f; a.v=v;
 return for_reduce(i,n,0,
     get,(void *)&a,sum,NULL);
}
int for_pos_next(int i, int n, int v,
 int (*f)()){
 Arg a; a.f=f; a.v=v;
 return for_reduce(i,n,0,
     get_next,(void *)&a,sum,NULL);
}
```

```
int sum(int i, int j,
        int a, int b, void *arg){
 return a+b;
}
void element_body(int i, void *arg){
 Arg *a=(Arg *)arg;
 int p, pn, v=a->f(i);
 p=for_pos(0,SIZE,i,a->f);
 pn=for_pos_next(0,SIZE,i,a->f);
 if (p<=a->i && a->i<pn) a->v=v;
}
int element_at_pos(int i){
 Arg a; a.f=my_random; a.i=i;
 for_loop(0,SIZE,element_body,
          (void *)&a);
 return a.v;
}
void sort_body(int i, void *arg){
 printf("%d ",element_at_pos(i));
}
```

Figure 18. The parallel sort



DOI: 10.1002/cpe

The initial set print launches 11 threads, among which 8 read one or two elements from the $my\_random$ function, i.e. copy a value from the code to the output buffer.

Figure 18 shows the sort. Function $element\_at\_pos(i)$ returns element at position $i$ in the sorted set. Function $for\_pos(..., v)$ returns the position of the first occurrence of $v$ in the sorted set. Function $for\_pos\_next(..., v)$ returns the position of the first element next to $v$ in the sorted set.

It implements a very poor sequential sorting algorithm, requiring O($n^3$) comparisons. For each element $e$ we recompute $n$ times the number of elements less than $e$ instead of building an array once and use it. In a parallel processor, such recomputations are faster than storing and loading a data structure.

The second *for_loop* in $main$ launches 11 threads, among which 8 compute one or two elements at their position in the sorted set.

The parallel threads are ordered when they are dynamically created, which sets the prints order. Each computed element in the output only uses the initial values, given by calls to $my\_random$. Element computations are all fully independent. Each is computed from a tree of threads having only depth dependences, thanks to the *for* reductions avoiding recurrences in the loop bodies.

To summarize, sorting a set of scalars computes in parallel all the elements of the sorted output. The duplicated elements at ranks $i$ to $j$ ($i \leq j$) are the ones in the input set which have $i$ lower and $j$ lower or equal elements. The sorting function does not use any array, any storage to permute. The complexity is O($(log\ n)^3$), i.e. all the threads are deployed after $(log\ n)^3$ steps (to be compared to the O($n * log\ n$) complexity of a sequential sorting algorithm).

## 4. COMPARING OS PARALLELIZATION TO HARDWARE PARALLELIZATION

Figure 19 shows a *pthread* implementation of the sum reduction. This code is compared to the one on figure 3. A first difference is that the *pthread* version explicits the parallelization through the calls to *pthread_create*, the synchronization through the calls to *pthread_join* and the communications through the arguments transmission at thread creation and the result transmission at thread exit and join. These explicit calls obscure the code: the *pthread* version is four times longer than the figure 3 version parallelized by our proposed hardware.

```
typedef struct{int *v; int n;} ST;
void *sum(void *st){
 ST s1,s2; long *s,*sl,*sr; pthread_t t1, t2;
 s =malloc(sizeof(long));
 sl=malloc(sizeof(long)); sr=malloc(sizeof(long));
 if (((ST *)st)->n>2){
  s1.v=((ST *)st)->v; s1.n=((ST *)st)->n/2;
  pthread_create(&t1,NULL, sum,(void *)&s1);
  s2.v=((ST *)st)->v + ((ST *)st)->n/2;
  s2.n=((ST *)st)->n - ((ST *)st)->n/2;
  pthread_create(&t2,NULL, sum,(void *)&s2);
  pthread_join(t1,(void **)&sl);
  pthread_join(t2,(void **)&sr);
 }
 else if (((ST *)st)->n==1){*sl=((ST *)st)->v[0];*sr=0;}
 else{*sl=((ST *)st)->v[0]; *sr=((ST *)st)->v[1];}
 *s=*sl+*sr; free(sl); free(sr);
 pthread_exit((void *)s);
}
```

```
#define SIZE 10
int v[SIZE]=
 {0,1,2,3,4
 ,5,6,7,8,9};
main(){
 ST st;
 int *psum;
 pthread_t tid;
 st.v=v; st.n=SIZE;
 pthread_create(
  &tid, NULL, sum,
  (void *)&st);
 pthread_join(tid,
  (void **)&psum);
 printf("sum=%d\n",
  *psum);
 free(psum);
}
```

Figure 19. A *pthread* parallelization of the sum reduction

The $pthread$ run creates 12 threads, as does the hardware parallelization (Figure 5 shows 11 threads for the parallelization of $sum$, to which one printing thread is added).



DOI: 10.1002/cpe

| x86 instructions run by | create | join | exit |
|---|---|---|---|
| $pthread$ parallelization | 727-736 | 136-755 | 10821-10987 |
| hardware parallelization | 3-5 | 0 | 0 |

Table I. Number of x86 instructions run to create, join and exit threads

### *4.1. The compared architectural cost of parallelization*

A second difference is that in the $pthread$ run, the calls to $pthread\_create$, $pthread\_join$ and $pthread\_exit$ add a high overhead. The number of x86 instructions run by these calls can be measured using $pin$ [10]. Table I shows the overhead in the run of the *sum* code in figure 19. The measure was done on a Intel Core i7-4900MQ operated by Ubuntu 14.04. The *pthread* code is compiled with $gcc$ 4.8.4-2 ($-O3$ and $-static$ options) and libpthread-stubs0-dev0.3-4.

The $pthread\_create$ primitive ran 727 (call in $main$) or 736 (calls in $sum$) x86 instructions. The $pthread\_join$ primitive ran 755 (call in $main$), from 143 to 736 (first call in $sum$) and from 136 to 755 (second call in $sum$) instructions. The $pthread\_exit$ primitive ran from 10821 to 10987 instructions (it is not clear what $pin$ exactly measures in $pthread\_exit$: 10K instructions run seems a lot; the measures for $pthread\_create$ and $pthread\_join$ have been confirmed in a second experience using $gdb$, which was not possible for $pthread\_exit$).

The last line of the table gives in contrast the very low number of x86 instructions run when the parallelization is done by hardware.

The hardware parallelization creates threads and sends registers initializations messages. In the $sum$ run (see figure 4), the first $sum$ call runs 5 instructions to fork, copy registers $rdi$, $rsi$ and $rbx$ and call. The second call runs 3 instructions to fork, copy register $rcx$ and call. The cost to synchronize threads is null because they are ordered and hardware register renaming offers a free natural synchronization between any reader and its unique writer.

The OS overhead condamns OS-based parallelization to coarse grain. To amortize 1.5K instruction run (or 12K if the $pthread\_exit$ cost is included), each thread should at least sum up a thousand values (resp. 12K). Parallelizing hardware makes fine grain parallelization possible: one thread per pair of values.

### *4.2. The compared microarchitectural cost of parallelization*

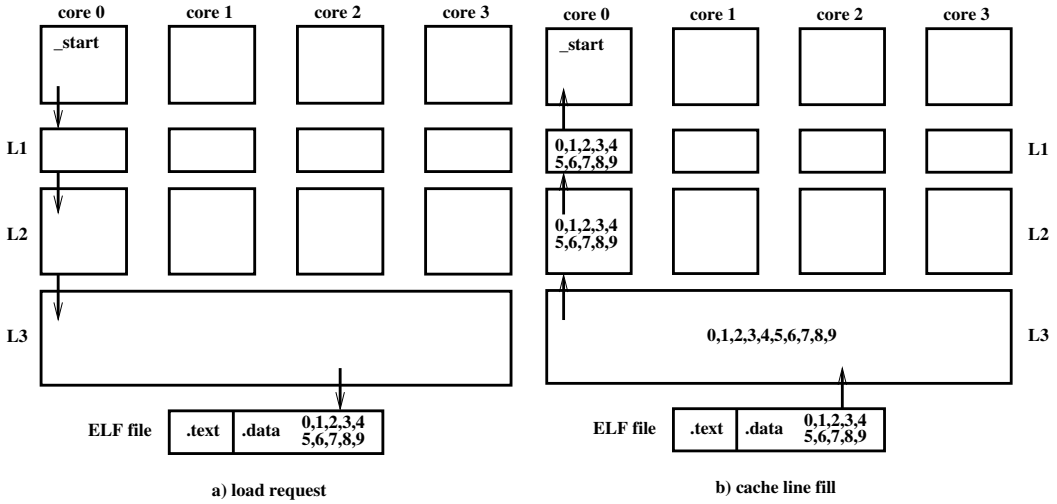


Figure 20. Data initializations for the $sum$ execution



DOI: 10.1002/cpe

The number of instructions run is one part of the cost of parallelization. A second part is the data movement from source input to result output, i.e. a microarchitectural cost.

In the $pthread$ program, the input vector is initialized from the ELF file by the OS $\_start$ function (which later calls $main$). The data are centralized in the L1 data cache of the core running the $\_start$ function. This is illustrated on figure 20. The left figure is the $\_start$ function architectural load request and the right figure is the memory hierarchy microarchitectural fill request.

Each leaf $sum$ thread reads one or two elements of summed vector $v$. These accesses trigger communications between the $v$ elements central location and each requesting core, as illustrated on figure 21. This is hardware driven by copying cache lines from L3 to L1. Bus contention may serialize the requests, i.e. the threads. Each copy caches a full 64 bytes line when one or two elements only are useful, uselessly transfering from 56 to 60 bytes in each communication.

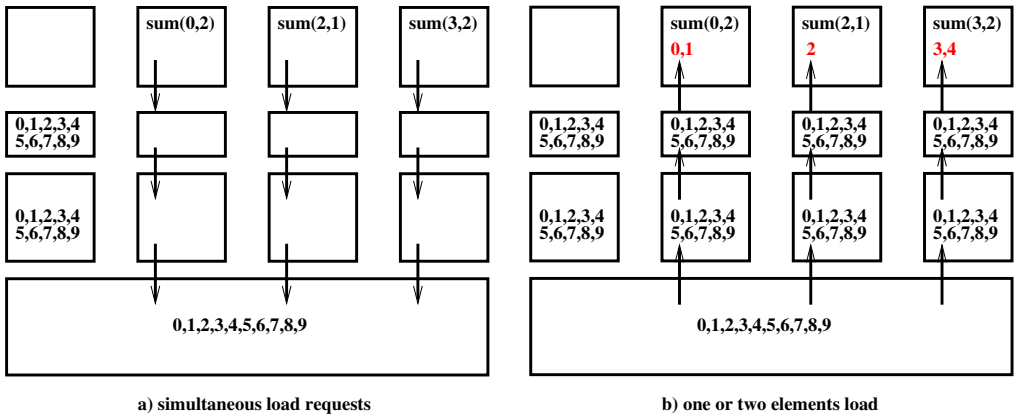


Figure 21. Data movements during the $sum$ execution parallelized by $pthread$

Caches are not adequate devices for parallel executions. The principle of spatial locality enforces caches to keep large lines of data, i.e. centralizing them. This is in conflict with data distribution. The principle of temporal locality enforces caches to keep data for multiple successive local accesses. This is in conflict with data recomputation which avoids storing.

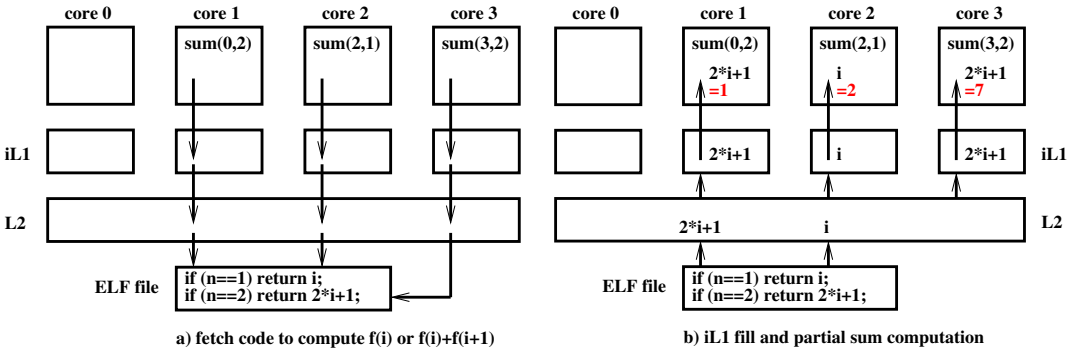


Figure 22. Data movements during the $sum$ execution parallelized by hardware

Figure 22 shows the data movements when running the code in figure 4 in parallel on a parallelizing hardware. Each leaf $sum$ thread computes its partial sum from values encoded in the fetched instructions. The computing code is read from instruction cache iL1 whereas in the $pthread$ run, data are read from data cache dL1 which holds vector $v$. The communications are reduced to the minimum, i.e. the values needed by the computation migrate from the code file to the cores.



DOI: 10.1002/cpe

*4.3. Comparing a data structure based parallelization to a function based one*

Figure 23 shows a $pthread$ version of $quicksort$. It is compared to figures 17 and 18 sort program.

```
void *quicksort(void *sa){
 pthread_t tid1, tid2; sub_array_t sal, sar;
 int ip, p, t, i1, i2, f, l;
 f=((sub_array_t *)sa)->f;
 l=((sub_array_t *)sa)->l;
 if (f < l){
  ip=f; i1=f; i2=l; p=a[ip];
  while (1){
   while (i1<l && a[i1] <= p) i1++;
   while (a[i2] > p) i2--;
   if (i1<i2){t=a[i1]; a[i1]=a[i2]; a[i2]=t;}
   else break;
  }
  a[ip]=a[i2]; a[i2]=p; sal.f=f; sal.l=i2-1;
  pthread_create(&tid1,NULL,
    quicksort,(void *)&sal);
  sar.f=i2+1; sar.l=l;
  pthread_create(&tid2,NULL,
    quicksort,(void *)&sar);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
 }
}
```

```
#define SIZE 10
typedef struct
      str {int f; int l;}
      sub_array_t;
int a[SIZE]={3,0,1,5,9,
             7,6,3,4,1};
void main(){
 int i;
 sub_array_t sa;
 pthread_t tid;
 for (i=0;i<SIZE;i++)
   printf("%d ",a[i]);
 printf("\n");
 sa.f=0;
 sa.l=SIZE-1;
 pthread_create(&tid,NULL,
   quicksort,(void *)&sa);
 pthread_join(tid,NULL);
 for (i=0;i<SIZE;i++)
   printf("%d ",a[i]);
 printf("\n");
}
```

Figure 23. A *pthread* parallelization of quicksort

Figure 24, 25 and 26 show how the data travel in the caches when the *pthread* quicksort function is run. The $\_start$ function copies the initialized vector from the ELF file into the core 0 memory hierarchy. The loop to print the initial vector belongs to the same thread as the $\_start$ function. It is run on core 0 and it accesses to the vector elements in cache L1.

The main thread creates a first $quicksort$ thread run on core 1. It gets the vector from L3 in its L1 to partition it (figure 24 left part: cache miss propagates; figure 24 right part: cache hierarchy load).

The partitioning *while(1)* loop updates the vector copy in the core 1 L1 cache (figure 25 left part). The $quicksort$ thread creates two new $quicksort$ threads, each to sort a half vector. The left half sorting thread is run on core 2 and the right half sorting thread is run on core 3. Both threads read their vector half from the core 1 L1, which has the only updated copy of the vector (figure 25 right part). Both requests have to be serialized.

On the left part of figure 26, we assume core 2 gets access first. The right part is core 3 access.

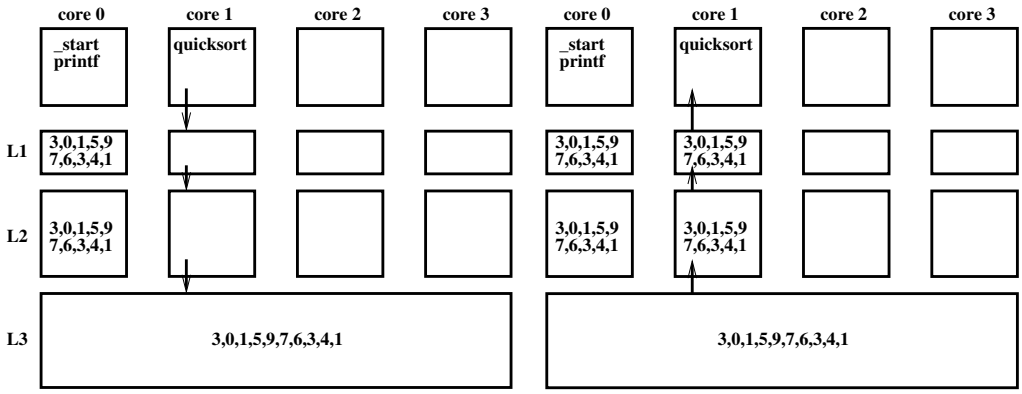


Figure 24. Data movements during the $quicksort$ execution parallelized by $pthread$



DOI: 10.1002/cpe

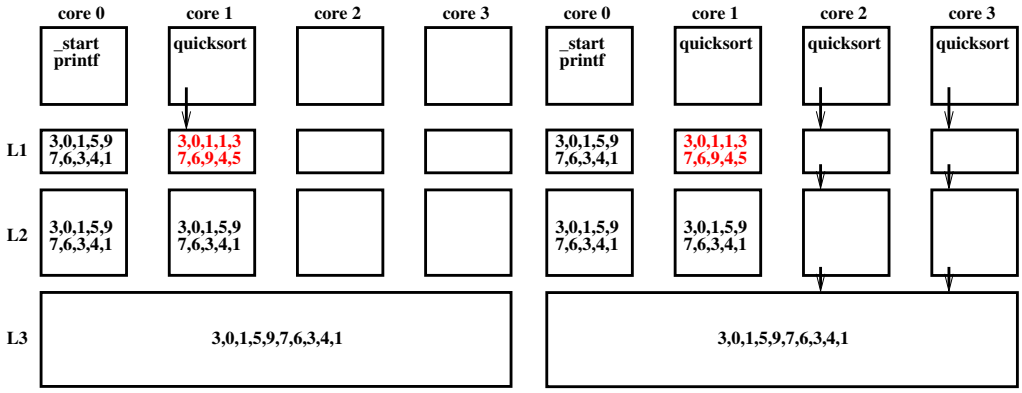


Figure 25. Data movements during the $quicksort$ execution parallelized by $pthread$

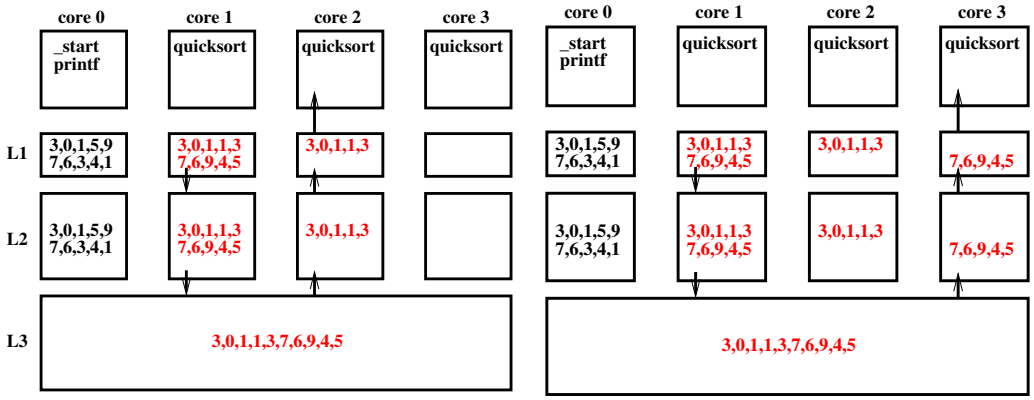


Figure 26. Data movements during the $quicksort$ execution parallelized by $pthread$

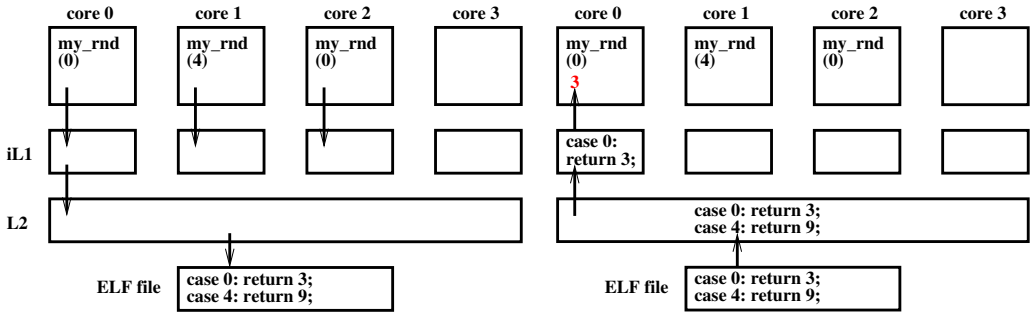


Figure 27. Data movements during the $sort$ execution parallelized by hardware

The data travel from the partitioning core to the partitions sorting ones. Each level of the $quicksort$ binary tree moves the full vector. There are $n * log\ n$ data movements from L1 to L1 through L2 and L3, with no locality benefit. All these movements are avoided in the sort program on figures 17 and 18. As the values to be sorted are held in the code, i.e. encoded in the machine instructions translating function $my\_random$, the only data movements come from the instruction memory hierarchy. Each core reads the data it needs from the closest instruction cache holding it. On figure 27, three threads run function $my\_random$ on three cores (abbreviated as $my\_rnd$). The three threads request the same portion of code from their L1 instruction cache (figure 27, left part). The three requests to L2 are serialized and it is assumed core 0 is served first. The requested code is loaded in the shared L2 (figure 27, right part).



DOI: 10.1002/cpe

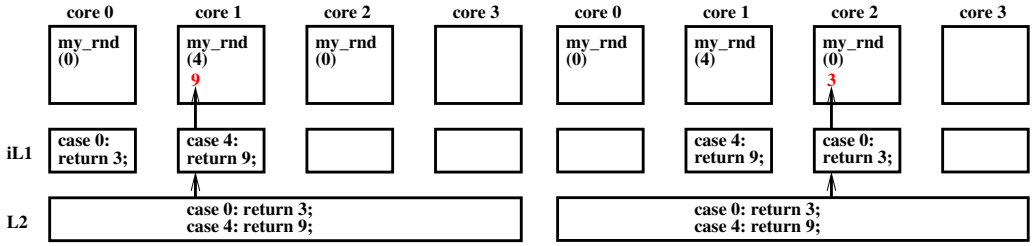


Figure 28. Data movements during the $sort$ execution parallelized by hardware

The next core to be served is core 1 (figure 28, left part). The memory hierarchy plays a role and L2 hits. Core 2 is the last to be served, directly from L2 (figure 28, right part).

## 5. RELATED WORKS AND CONCLUSION

Parallel programming concerns automatic parallelization (i.e. by the compiler) and "hand-made" parallelization using APIs (Pthreads, MPI, OpenMP for CPUs and CUDA, OpenCL for CPUs+GPUs). In 2012 a survey was published of parallel programming models and tools for multicore and manycore processors [11].

Automatic parallelization parallelizes loops with techniques based on the polyhedral model [12]. If some addresses in a loop are dynamic (e.g. pointer-based), the compiler cannot optimally parallelize. It is also the case if the loop control is complex or if the iteration body contains statically unknown control (e.g. involving special exits via return or break instructions). In [13], some transformation techniques are added to the polyhedral model to remove some of these irregularities.

The model we propose assumes that part of the parallelization is hand-made, i.e. structuring the program with template functions replacing all the *for* and *while* loops with their functional divide-and-conquer equivalents. Anything else is taken in charge by the hardware without any compiler, library or OS primitive intervention.

Parallelization based on OS threads [14] [15] suffers from the overhead of OS primitives, the opacity of the code which must exhibit the synchronizations and communications and its dependency on the number of cores through the explicit creation of threads. A major drawback of OS threads is their non deterministic behaviour, as pointed out by Lee [16].

The parallelizing hardware we propose has low architectural overcost (a few machine instructions run at thread creation, compared to thousands of instructions run in the *pthread* API).

The existing contributions on a hardware approach to automatize parallelization [17][18][19][20] are penalized by the low basic Instruction Level Parallelism (ILP) measured in programs [21]. The hardware based parallelization in [22] overcomes this limitation in 3 ways: (i) very distant ILP is caught when fetch is parallelized, (ii) all false dependences are removed through full renaming and (iii) many true dependences are removed by copying values. The remaining dependences in a run are true ones related to algorithmic sequentialities that the program implements. In such conditions, the authors in [23] have reached a high ILP (thousands), increasing with the data size, on the parallel benchmarks of the PBBS suite [24].

In the hardware design we propose, fetch is parallelized and there is no data memory, i.e. no memory dependences. In such conditions high ILP can be captured in a program run when the program implements a parallel algorithm. The microarchitectural cost of parallelization is reduced because there are less communications, involving only neighbour cores. The proposed programming style avoids data storing which simplifies the hardware, makes parallel computations more independent and uses well-known functional programming paradigm as in parallel Haskell [25]. Instead of computing a data structure globally, we compute its elements individually and in parallel. The efficiency does not rely on the cache locality principle, which applies poorly to a parallel run. Instead, it relies on parallel locality, as defined in [26].



DOI: 10.1002/cpe

The number of transistors on a chip allows the integration of thousands of simple cores such as the design proposed in this paper. Parallelization should be done fastly and reliably, leading to reproducible computations as the programming model proposed in this paper.

## REFERENCES


1. Mittal S, Vetter JS. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.* Jul 2015; **47**(4):69:1–69:35.
2. Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: Maximizing on-chip parallelism. *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, ACM: New York, NY, USA, 1998; 533–544.
3. Keller RM. Look-ahead processors. *ACM Comput. Surv.* Dec 1975; **7**(4):177–195.
4. Gropp W, Thakur R, Lusk E. *Using MPI-2: Advanced Features of the Message Passing Interface*. 2nd edn., MIT Press: Cambridge, MA, USA, 1999.
5. Hudak P. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.* Sep 1989; **21**(3):359–411.
6. Hartel P, Muller H, Glaser H. The functional “c” experience. *J. Funct. Program.* Mar 2004; **14**(2):129–135.
7. Hudak P, Hughes J, Peyton Jones S, Wadler P. A history of haskell: Being lazy with class. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, ACM: New York, NY, USA, 2007; 12–1–12–55.
8. Barendregt HP. *Handbook of Theoretical Computer Science (Vol. B)*. MIT Press: Cambridge, MA, USA, 1990.
9. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, USENIX Association: Berkeley, CA, USA, 2004; 10–10.
10. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.* 2005; **40**(6):190–200.
11. Diaz J, Munoz-Caro C, Nino A. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on* 2012; **23**(8):1369–1386.
12. Feautrier P. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 1991; **20**(1):23–53.
13. Benabderrahmane MW, Pouchet LN, Cohen A, Bastoul C. The polyhedral model is more widely applicable than you think. *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, Springer-Verlag: Berlin, Heidelberg, 2010; 283–303.
14. Chapman B, Jost G, Pas Rvd. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
15. Butenhof D. *Programming with POSIX Threads*. Addison-Wesley, 1997.
16. Lee EA. The problem with threads. *Computer* May 2006; **39**(5):33–42.
17. Kim HS, Smith J. An instruction set and microarchitecture for instruction level distributed processing. *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002; 71–81.
18. Mehrara M, Hao J, Hsu PC, Mahlke S. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM: New York, NY, USA, 2009; 166–176.
19. Ranjan R, Latorre F, Marcuello P, Gonzalez A. Fg-stp: Fine-grain single thread partitioning on multicores. *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011; 15–24.
20. Sharafeddine M, Jothi K, Akkary H. Disjoint out-of-order execution processor. *Transactions on Architecture and Code Optimization (TACO)* sept 2012; **9**(3):19:1–19:32.
21. Wall DW. Limits of instruction-level parallelism. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, 1991; 176–188.
22. Goossens B, Parello D, Porada K, Rahmoune D. Toward a core design to distribute an execution on a manycore processor. *Parallel Computing Technologies*, *Lecture Notes in Computer Science*, vol. 9251, Malyshkin V (ed.). Springer International Publishing, 2015; 390–404.
23. Goossens B, Parello D. Limits of instruction-level parallelism capture. *Procedia Computer Science* 2013; **18**(0):1664–1673. 2013 International Conference on Computational Science.
24. Shun J, Blelloch GE, Fineman JT, Gibbons PB, Kyrola A, Simhadri HV, Tangwongsan K. Brief announcement: The problem based benchmark suite. *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, 2012; 68–70.
25. Marlow S, Peyton Jones S, Singh S. Runtime support for multicore haskell. *SIGPLAN Not.* Aug 2009; **44**(9):65–78.
26. Goossens B, Parello D, Porada K, Rahmoune D. Parallel locality and parallelization quality. *7th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2016.




DOI: 10.1002/cpe