Performance and Overhead Measurements on the Makbilan

Yosi Ben-Asher   Dror G. Feitelson
Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
E-mail: {yosi,drorf}@cs.huji.ac.il

Technical Report 91-5

October 1991

Abstract

The Makbilan is a research multiprocessor with 12 single-board computers in a Multibus-II cage. Each single-board computer is a 386 processor running at 20 MHz with 4 MB of memory. The software includes a local kernel (Intel's RMK) on each board, and a parallel run-time library that supports the constructs of the *ParC* language. We report measurements of bus contention effects, context-switch overhead, overhead for spawning new tasks, the effectiveness of load balancing, and synchronization overhead on this system.

1 Introduction

When evaluating a new system, it is important to make accurate measurements of the overheads involved in various system operations. This is not always as easy as it sounds.

On the Makbilan, and bus-based shared memory machines in general, the problem is one of isolating the measurements from irrelevant external effects. For example, the measured results depend on the following:

- Whether data structures are local or remote, and must be accessed through the bus.
- If remote accesses are made, what is the contention for the bus at that time.
- What is the contention for a specific memory module. This influences both local and remote references.
- What is the load on the PEs, i.e. how many tasks are being time shared.

The methodology used is to write application programs that perform a certain operation a large number of times, and measure their execution times. The applications are written so that the transient effects at startup are negligible relative to the total execution time, thus enabling us to attribute the whole time to the repeated performance of the operation being measured. However, care must be taken that some additional operation does not "slip in" together with the one we are interested in.

Hardware Configuration

The Hardware configuration of the Makbilan, as used in these measurements, was as follows:

- A Multibus-II backplane with 20 slots.
- A CSM-001 bus controller.
- 12 single-board computers (SBC) acting as PEs. Each has a 386 processor running at 20 MHz, a 387 coprocessor, an MPC, and 4 MB of memory.
- Another SBC acting as a Unix host.
- A 186/410 board with six terminal ports.
- A 186/224 board peripheral controller interfacing the bus with the SYP 500 chassis. This chassis contains the I/O devices (disk and tape).

2 Processing Rates

Each processor in the Makbilan has an independent clock. The first experiment is designed to check how well these clocks are synchronized. The code used is as follows:

```
lparfor int n; 0; proc_no-1; -1; {
   addr[n] = (int*) malloc( sizeof(int) );
} epar
for (i=0 ; i<proc_no ; i++) {
   for (j=i+1 ; j<proc_no ; j++) {
      for (k=0 ; k<EXPR ; k++) {
         lparfor int n; 0; proc_no-1; -1;
         {
            int *loc, rem, it1, it2;
            if ((n != i) && (n != j))
               pcontinue; /* check i against j */
            it1 = ITER1;
            it2 = ITER2;
            loc = addr[n];
            *loc = 0;
            if (n == i) {
               while (*loc < it2) {
                  if (*loc == it1)
                     rem = *(addr[j]);
                  (*loc)++;
               }
               res[i][j][k] = rem;
            }
            else {
               while (*loc < it2) {
                  if (*loc == it1)
                     rem = *(addr[i]);
                  (*loc)++;
               }
               res[j][i][k] = rem;
            }
         } epar
      }
   }
}
```

All this code does is to check all pairs of processors against each other. this is done by spawning a pair of activities, one on each processor, which iterate a large number of times — 12,000,000 was used. A bit before the end, at 10,000,000 iterations, each activity checks the progress of the other activity and tabulates it. Everything is local with no interference from any other processor in the system.

The results are shown in table 1. These are averages of five measurements for each pair. Row i of the table contains the rates of all the processors relative to processor number i. Thus a row with many positive numbers, e.g. row 3, indicates a relatively slow processor. A row with many negative numbers, e.g. row 7, indicates a relatively fast processor. The greatest difference measured,

	1	2	3	4	5	6	7	8	9	10	11	12
1	—	+7	−54	+2	0	+25	+19	+1	+2	−9	−6	−10
2	−7	—	−60	−7	+2	+18	+18	−3	−5	−16	0	−8
3	+54	+60	—	+59	+53	+79	+80	+45	+54	+42	+59	+45
4	−2	+8	−58	—	0	+29	+25	+5	−1	−4	−3	−11
5	0	−1	−53	0	—	+23	+15	+11	−17	−19	+4	−12
6	−25	−17	−80	−24	−23	—	−10	−35	−28	−45	−20	−45
7	−18	−17	−80	−25	−15	+10	—	−22	−30	−34	−24	−34
8	−1	+3	−45	−4	−11	+36	+22	—	+5	−20	+5	−10
9	−2	+5	−54	+1	+18	+28	+30	−5	—	−5	+7	+2
10	+9	+17	−41	+4	+19	+45	+34	+20	+5	—	+9	−5
11	+6	0	−59	+3	−4	+20	+24	−5	−6	−9	—	−17
12	+10	+8	−44	+12	+12	+45	+34	+11	−2	+5	+17	—

Table 1: *Relative rates of the 12 processors currently in the Makbilan, in parts per million.*

between processor 3 and processor 6, is only 80 parts per million. At least the last digit of these measurements should not be considered as accurate. For example, the table shows that processor #8 is slower than #11 by 5 ppm, and processor #4 is slower than #8 by an additional 4 ppm, but processor #4 is faster than #11 by 3 ppm.

3 Bus Contention

The Makbilan backplane is a Multibus-II, with a throughput of 40MB/s. This seems to be more than enough to service a number of transfers within one processor cycle, implying that the bus should not be a bottleneck. But the Makbilan uses the bus mainly for remote memory access, and the bus protocol requires that the bus be held until the transaction terminates. The memory has dual ports, and access from the bus may suffer several wait states. Thus the memory response time may be a bottleneck that limits the bus throughput. The experiments in this section are designed to check this.

3.1 Contention for Bus Access

This subsection contains two experiments: the first shows how performance degrades when more processors try to use the bus simultaneously, and the second shows how the mix of local and global accesses influences the performance.

Experiment 1

This experiment is repeated with different numbers of processors. The code for each run is as follows:

```
int g;
lparfor int j; 1; num-of-procs; -1;
{
    int k;
    /*register*/ int *pg, lg;
    pg = &g;
    for (k=0 ; k<ITER ; k++) {
        *pg=lg; /* repeated 100 times */ *pg=lg;
    }
} epar
```

The assignment in the loop is repeated 100 times in-line to reduce the effect of the loop handling itself. The experiment was conducted twice, once with `pg` and `lg` defined as local variables and once as registers. Note that the global variable `g` is actually in the local memory of the first processor, so in the measurement for i processors only $i-1$ are really using the bus. In particular, the measurement for one processor is really a local access, not a global one.

The results are shown in fig. 1. It is obvious that a linear relationship exists between the number of processors and the elapsed time, indicating that the bus cannot support multiple requests within a single processor cycle. Thus the bus is indeed a bottleneck. As may be expected, when the

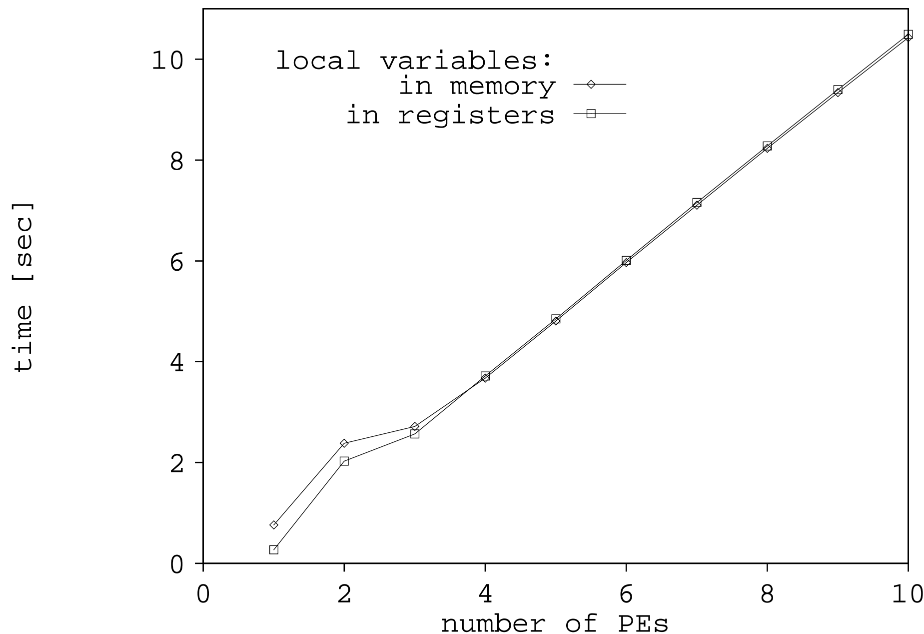

Figure 1: *Measurement showing that the bus is not faster than the processors. Time for 1000000 global accesses from each processor is shown.*

local variables are in registers the elapsed time is slightly shorter. Note, however, that as more processors are added the difference decreases until it disappears. The reason for this is that using registers reduces the time needed to issue the next global access, but the bottleneck is the global access itself. Issuing the instruction faster does not help.

Experiment 2

The load on the bus depends, of course, on the access rate from the different processors. This in turn depends on the ratio of local to global memory references. This experiment shows how a larger percentage of local references makes the bus seem faster.

The basic code is similar to the previous experiment, except that each processor does global accesses to a different global variable. The global variables are arranged so that each is in the memory of a different processor. This ensures that the bottleneck is indeed the bus (and its protocol), and not any overloading effects on a specific memory module.

```
/* allocate memory in every processor */
lparfor int j; 1; proc_no; -1;
    M[j] = (int *) malloc(sizeof(int));
epar
/* repeat for different numbers of PEs */
lparfor int j; 1; num-of-procs; -1;
{
```

```
    int l, k, *p;
    /* arrange pointers */
    if (get_pid() < proc_no)
        p = M[get_pid()+1];
    else
        p = M[1];
    /* perform local and global accesses */
    for (k=0 ; k<ITER ; k++) {
        DOx( *p, l )
    }
} epar
```

The DOx macro includes 100 instructions, out of which x are assignments to the global variable, of the form `*p=l`, and the rest are increments of the local variable, i.e. `l++`. results for five values of x are reported: 1, 10, 20, 50, and 100. for example, the macro `DO50(x,y)` looks like this:

```
#define DO50(x,y) { \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; \
l++; x=y; l++; x=y; l++; x=y; l++; x=y; l++; x=y; }
```

As may be expected, when practically all the accesses are local, the number of processors working in parallel has no influence ($x = 1$). When a certain fraction of accesses are global, bus contention begins to take its toll ($x = 50$ to $x = 50$), and the larger the fraction of global accesses the larger the elapsed time becomes ($x = 100$).

The results for heavy bus activity are practically identical to those from experiment 1. this means that it is not possible to decouple the degradation due to bus contention from the degradation due to contention for the same memory module.

3.2 Bus Access Priorities

The Multibus-II arbitration mechanism gives higher priority to PEs with lower serial numbers. This experiment was designed to check how big the difference in performance can be. The code is:

```
/* allocate a counter in every processor */
lparfor int j; 1; proc_no; -1;
{
    C[j] = (int *) malloc(sizeof(int));
    *(C[j]) = 0;
```

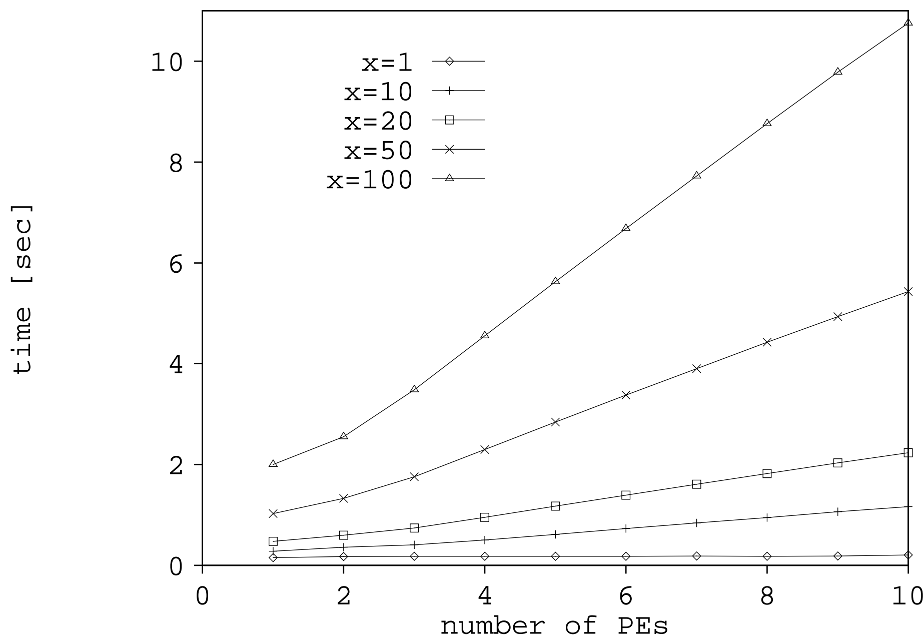

Figure 2: *Effect of different ratios of global to local accesses on the degradation due to bus contention.*

```
} epar
flag = 1;
/* increment neighbor's counter until threshold reached */
lparfor int j; 1; proc_no; -1;
{
    int k, *p;
    p = C[(j<proc_no)?  j+1 :  1];
    while (g) {
        for (k=0 ; k<ITER ; k++)
            *p = *p + 1;
        if (*p >= THRESHOLD)
            g = 0;
    }
} epar
```

Again, each processor increments a counter in another processor's local memory. Note that the accesses are done in a double loop: the outer loop checks the termination condition (when the first PE reaches the threshold value), and the inner one performs a certain constant number of iterations. This is done to reduce the effect of checking the termination condition.

The results are tabulated in table 2. The relative performance displays a step-like behavior. Using PE number 1 as a reference point, we find that PE number 2 achieves the same number of accesses, PE number 3 achieves 90% of that, PEs numbers 4 through 6 achieve about 2/3, and

serial number	*number of accesses*
1	1000
2	993
3	893
4	688
5	671
6	658
7	501
8	501
9	501
10	501

Table 2: *Number of accesses performed by different PEs, when all compete for the bus.*

PEs 7 through 10 achieve only 1/2. The reason for this behavior is that the arbitration mechanism operates in a gated manner. First, all the contending PEs are noted. Then they are serviced according to their serial numbers. While this is being done, new requests that arrive are blocked out. Only when the previous set have all been serviced, does a new round begin. Thus PEs with low serial numbers, that are serviced early in the round, manage to generate a new request before the round is over, and therefore get into the next round as well. PEs late in line are serviced towards the end of the round, and do not manage to generate a new request before the round is over. Therefore they miss alternate rounds. PEs in the middle sometimes make it and sometimes not.

4 Remote Access Penalty

Obviously a remote access takes more time than a local one. But how much more? The experiments in this section are designed to answer this question, and characterize the performance in different operating conditions.

4.1 Experiments

All these experiments have the same structure. A set of activities is spawned and access some variables in a loop. In each set there is always one activity that accesses a local variable, such that no other activity accesses any variable in its local memory. This activity is therefore isolated from all the others, and serves as a reference point. When it completes 100000 iterations, it notes the number of iterations that the other activities have completed in the same time. This gives the relative execution rates. Of course the measured time is for the whole instruction execution cycle, not only the memory reference to get the operands. To reduce this effect, the instruction that is executed is the `increment` instruction, which performs an increment on a memory address. This requires two memory accesses, and only minimal decoding. In addition, the overhead for executing the loop itself (which is local) may obviously bias the performance figures. To reduce this effect,

the loop contains in-line code for 250 consecutive increment instructions. The code for the first experiment is:

```
pparblock
{ /* local undisturbed */
    register int  *p;
    int           i, res2, res3;
    p = &i;
    while (flag1 == 0)
        ;
    for ((*p)=0 ; (*p)<ITER ; ) {
        (*p)++; /* repeated 250 times */ (*p)++;
    }
    res2 = *p2;
    res3 = *p3;
    sync;
} :
{ /* local disturbed */
    register int  *p;
    int           i, j;
    p3 = &j;
    flag2 = 1;
    p2 = &i;
    p = &i;
    while (flag1 == 0)
        ;
    for ((*p)=0 ; (*p)<ITER ; ) {
        (*p)++; /* repeated 250 times */ (*p)++;
    }
    sync;
} :
{ /* remote */
    register int  *p;
    while (flag2 == 0)
        ;
    p = p3;
    flag1 = 1;
    for ((*p)=0 ; (*p)<ITER ; ) {
        (*p)++; /* repeated 250 times */ (*p)++;
    }
    sync;
}
epar
```

access	*conditions*	*rate*
local	free memory	1.00
	contention from a single remote activity	0.74
	contention from multiple remote activities	0.57
remote	to unused memory, no bus contention	0.11
	to locally used memory, no bus contention	0.11
	with heavy bus contention	0.01–0.03

Table 3: *Relative access rates under different conditions.*

The variable `flag1` is used to signal that all are ready and the measurement can begin. `flag2` signals that the global pointer `p3` now points to a local variable, and can be used by the process that performs the remote accesses. `res2` and `res3` take a snapshot of the progress of the locally disturbed activity and the remote activity, respectively, at the instant that the locally undisturbed activity finishes its iterations.

The other experiments follow the same principle. To measure remote access to an unused memory, the middle (local disturbed) activity is terminated with a `pcontinue` after it allocates local memory and assigns its address to `p3`. `malloc` is used so that this memory will persist after the activity terminates. To measure contention from multiple activities, the third activity (remote) is replaced by a `pparfor` that spawns `proc_no - 2` additional activities, which all access the same variable.

4.2 Results

The results of these experiments are summarized in table 3. Remote references cause a degradation in the local access rate, and reduce it to 56–74% of the rate when there are no remote references. Under ideal conditions, remote references take nine times as long as local references to an unused memory, or 5–7 times as long as local references that are disturbed by remote references. Under less ideal conditions, where bus contention also takes its toll, remote references are much slower. The most extreme difference is between local references to an unused memory and remote references that suffer sever contention; in this case the remote references are from thirty to seventy times slower. The large variance is due to the fact that different PEs have different priorities, and therefore suffer different degrees of degradation under contention. Under reasonable conditions, where all the memories are accessed both locally and through the bus, and there is moderate contention, the ratio between local and remote references is about a factor of ten to twenty.

5 Scheduling

The scheduling overhead is the time required to perform a context switch. The main problem with measuring this quantity is that the run time library has a few tasks that compete with the user tasks for the CPU. This adds overhead that is unaccounted for.

5.1 The Default Library

This experiment measures the scheduling overhead by spawning a set of activities, one per PE. Each activity performs an empty loop, yielding the processor in each iteration. Thus the average time required per iteration is actually the time required to reschedule the activity. This scenario is repeated for different loads, so as to distinguish between the constant overhead due to the library tasks and the scheduling overhead that is proportional to the number of activities that are being time shared. The code used is the following:

```
lparfor int i; 1; proc_no; -1;
{
   start = get_loctime();
   for (j=0 ; j<ITER ; j++)
      {}
   stop = get_loctime();
   control[i] = (float) stop - start;
}
if (load == 1)
   sem_init( &ready, 1 );
else
   sem_init( &ready, 0 );
pparblock
{ /* create loading conditions */
   pparfor int l; 2; load; 1;
   {
      lparfor int i; 1; proc_no; -1;
      {
         int   j;
         if ((l == 2) && (i == 1))
            V( &ready ); /* signal when ready */
         for (j=0 ; j<ITER+load+3 ; j++) {
            KN_sleep( 0 );
         }
      }
      epar
   }
   epar
}:{ /* perform measurements */
   P( &ready ); /* wait for load to be ready */
   lparfor int i; 1; proc_no; -1;
   {
      int   j, start, stop;
      start = get_loctime();
      for (j=0 ; j<ITER ; j++) {
```

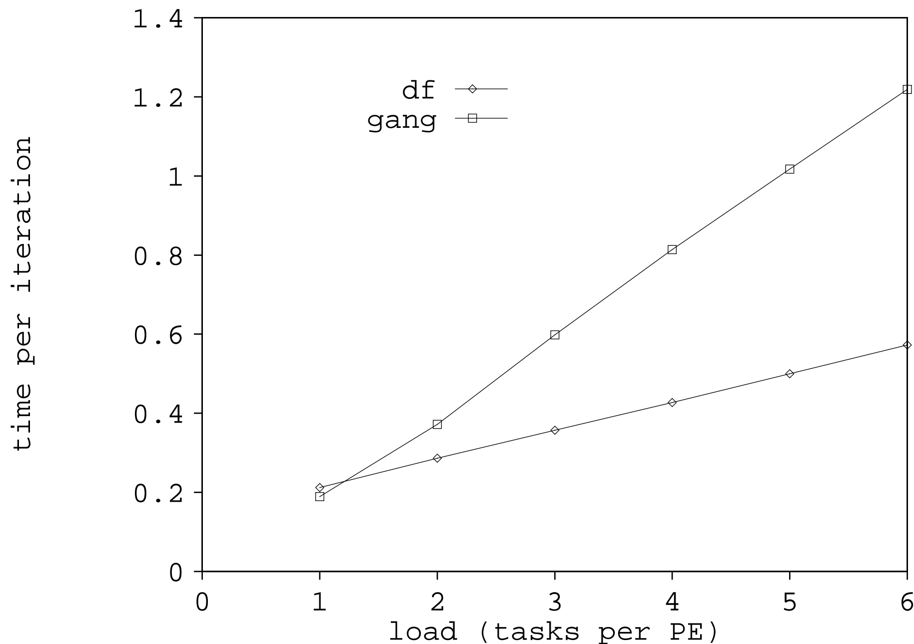

Figure 3: *Overhead for scheduling.*

```
            KN_sleep( 0 );
        }
        stop = get_loctime();
        for (j=0 ; j<CUSHION ; j++)
            ;
        res[i] = (float) stop - start;
        res[i] = res[i] - control[i];
    }
    epar
}
epar
```

The `control` measurement evaluates the time needed do the loop itself. The `ready` semaphore is used to trigger the measurement after the additional loading activities have been spawned. The `CUSHION` delays writing the results into the shared array, so as not to interfere with other measurements that have not terminated yet.

The results of this experiment are shown in figure 3. The average over 12 PEs was measured. As expected, the total scheduling overhead in a scheduling round is a linear function of the load, obeying the expression $0.140 + 0.072 \cdot load$ (where the measurement is in ms). This means that each RMK context switch takes about 72μs. The additional constant overhead of 140μs is attributed to the scheduling and execution of the `get_work` task. This overhead would be bigger if this task actually had something to do; 140μs is the time needed to find out that there is nothing to do.

It should be noted that the measurements show that the overhead on the master PE (PE #1) is between 2 and 4% higher than on the other PEs. This difference may be due to the fact that the `get_work` tasks on all the other PEs access global data structures that are located in the memory of the first PE, thus slowing it down.

5.2 The Gang Scheduling Library

This experiment is designed to measure the context switching overhead when the gang scheduling library is used. With this library, context switching is coordinated and done simultaneously on all the PEs, by means of a broadcast interrupt. To measure the overhead, a set of activities is created, one per PE. All the activities loop endlessly on a local variable, but one of them calls the library function `ts_sched` in each iteration; this function sends the scheduling broadcast. The code used is

```
int   flag=1;
pparfor int l; 1; load; 1;
{
   lparfor int i; 1; proc_no; -1;
   {
      int   j, start, stop;
      if (i == 2) { /* to avoid flag traffic */
         start = get_loctime();
         for (j=0 ; j<ITER ; j++) {
            ts_sched( 0 );
         }
         stop = get_loctime();
         flag = 0;
         res = (float) stop - start;
      }
      else {
         while(flag) {
            for (j=0 ; j<1000 ; j++)
               ;
         }
      }
   }
   epar
}
epar
```

The results of this experiment are also shown in figure 3. Again the overhead is more or less linear with the load, but in this case there is no added constant because there is no `get_work` task. However, the overhead is significantly higher than for the `df` library: it is approximately $0.2 \cdot load$. This is reasonable since every scheduling operation here involves two RMK scheduling operations, a broadcast, and the execution of the scheduler task.

6 Spawning new activities

The question here is "how much time does it take to create a new activity, execute it, and terminate it". This is important since activities that are too short relative to this overhead usually cause a degradation rather than an improvement in performance.

6.1 Experiment 1

In this experiment a set of empty activities is spawned, one per PE. This is repeated a large number of times, and the average time needed for a spawn in reported. The following code was used:

```
start = get_loctime();
for (j=0 ; j<lim ; j++)
   {}
stop = get_loctime();
control = (float) stop - start;

start = get_loctime();
for (j=0 ; j<lim ; j++) {
   lparfor int i; 1; proc_no; -1;
      {}
   epar
}
stop = get_loctime();
res = (float) stop - start;

time_per_spawn = (res - control)/lim;
```

Values of `lim` that were used were between 10000 and 50000. Rounded results were then typically the same to within less than 1%.

The sequence of events in each iteration, in the `df` library, is as follows:

1. The parent activity runs on the master PE, and
   (a) Starts another loop.
   (b) Spawns a new set of activities. This involves
      i. Initialization of the spawn descriptor.
      ii. Linking the descriptor to the global list.
      iii. Suspending execution.
2. The `get_work` task runs (on all PEs), doing
   (a) Get a lock on the global descriptor list and get a new activity.
   (b) Create a new RMK task to execute the activity.
   (c) Yield the processor.
3. The new activities execute on all the PEs. This includes

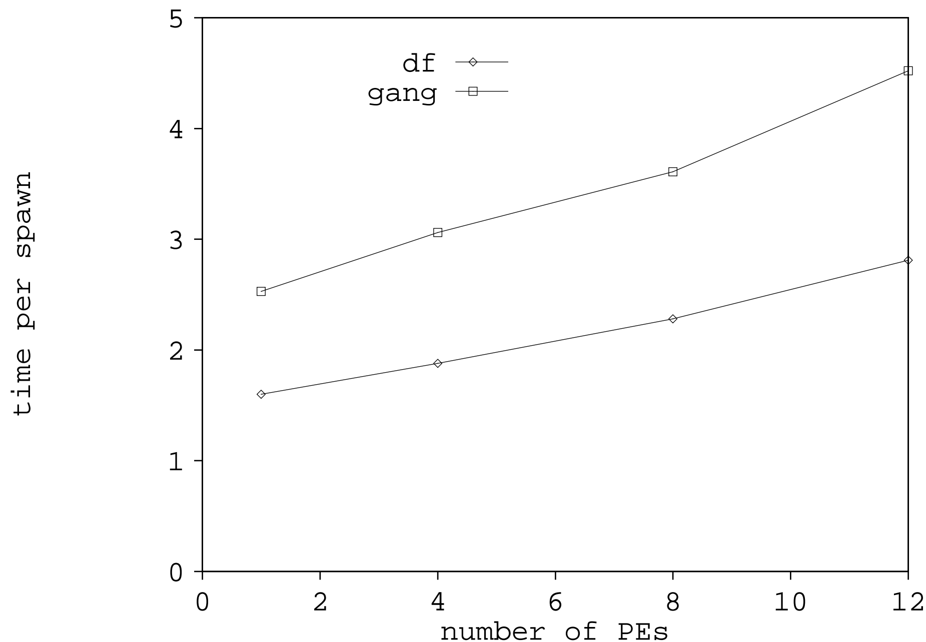

Figure 4: *Results for experiment 1: overhead for spawning empty activities on all the PEs.*

   (a) The selector runs and gets whatever is needed from the parent.
   (b) The activity terminates and the task is deleted. The last one links the parent's descriptor to the resume list.

4. the `get_work` task runs again (on the master PE), doing
   (a) Resume the parent.
   (b) yield the processor.

The results are shown in figure 4. These results indicate that as more PEs are used, the overhead increases. This increase is slightly superlinear, probably due to contention for the bus and for the spawn descriptor, and possibly also to the fact that the additional PEs have to perform remote accesses.

The increased overhead in the gang scheduling library is probably due to the fact that the library scheduler must run to perform all the necessary context switches.

6.2 Experiment 2

The main problem with the previous experiment is that it measures the total time to perform a parallel construct, not just the time to spawn an activity. To measure the net time required just to get an activity and execute it, without the time needed for the spawn, we need to have one PE that just creates activities all the time, while other PEs keep the global list of spawn descriptors from becoming empty. This is achieved by the following code:

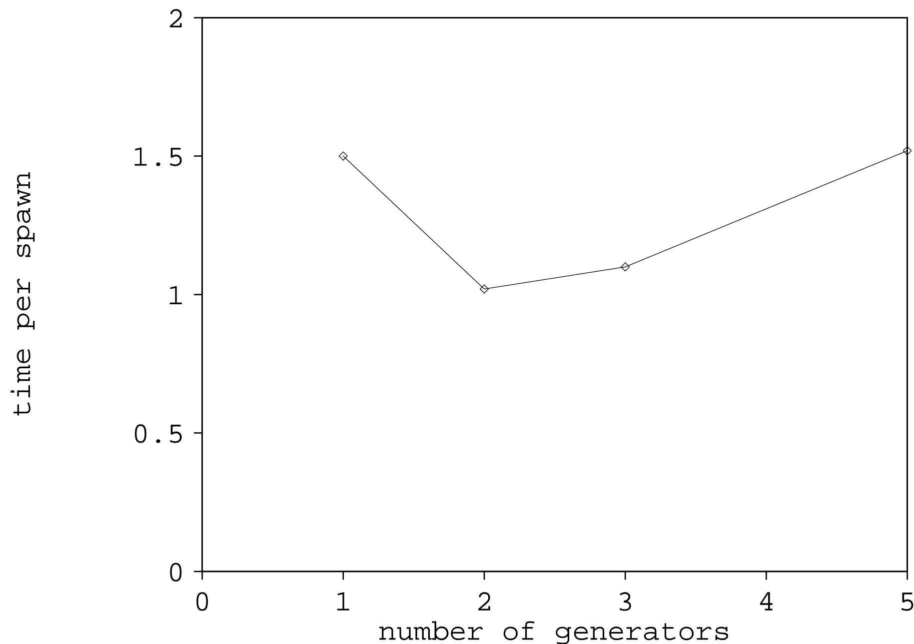

Figure 5: *Results for experiment 2: overhead for creating empty activities from a remote descriptor.*

```
start = get_loctime();
lparfor int j; 1; proc_no; -1;
{
   intk, lim;
   if (j == 1)
      pcontinue;             /* leave PE #1 free */
   lim = ITER;
   for (k=0 ; k<lim ; k++) { /* spawn single task on PE # 1 */
      lparfor int l; 1; 1; -1;
         {}
      epar
   }
}
epar
stop = get_loctime();
res = (float) stop - start;
time_per_spawn = res/(ITER*(proc_no-1));
```

The results are shown in figure 5. When only one activity generates new descriptors, the PE that creates the activities has to wait for the next one each time. When there are two or more, it does not have to wait, so the time per spawn is reduced. If there are too many generators, however, they contend for the global descriptor list, causing a degradation in performance.

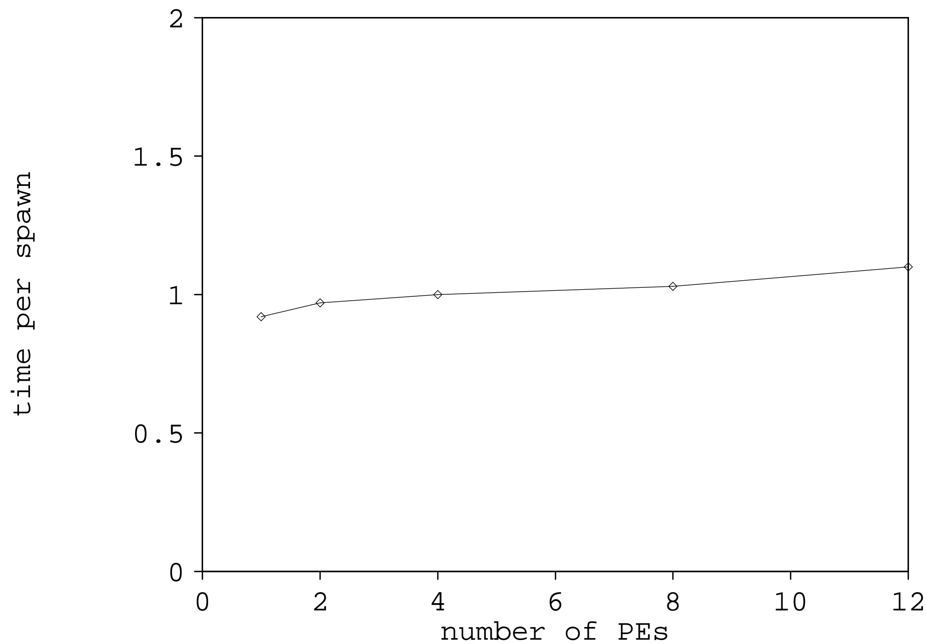

Figure 6: *Results for experiment 3: overhead for creating empty activities from a large descriptor.*

6.3 Experiment 3

As a final touch, we also consider a simpler version of the previous experiment. In this case a large number of activities are spawned at once, and executed by all the processors. Thus there is no contention for the global descriptor list due to the addition of new descriptors, but only contention due to the execution of activities. However, one of the PEs has the descriptor in its local memory, while the others have to access it via the bus. The code is:

```
lim = ITER*proc_no;
start = get_loctime();
pparfor int i; 1; lim; 1;
   {}
epar
stop = get_loctime();
res = (float) stop - start;
time_per_spawn = res/ITER;
```

The results of this experiment are shown in figure 6.

6.4 Summary

The experiments described in this section show that the effective overhead for spawning an activity is highly variable. The reason for this is that spawning is communication intensive, and hence

depends on the loading conditions. The minimum time needed just to create a task, start it and terminate it, is about 1ms. This number grows when there is contention for the bus, either due to other tasks being spawned, or due to computation.

7 Load Balancing and Granularity

The *ParC* run time library spawns tasks as follows: the activity that performs the spawn places a spawn descriptor in global memory, and each processor checks the descriptor and creates a new activity once every scheduling round. This should provide a measure of load balancing, because processors that are overloaded have a longer scheduling round, so they will take additional work at a slower rate.

It should be noted that load balancing can be defined in two ways: according to the numerical load and according to the work/performance load. Balancing the numerical load means that each processor will have the same number of activities. Balancing the work/performance means that each processor will do work proportional to its performance: inefficient processors will do less, so as not to cause a delay. The two are not the same, even if all the activities are identical, because the efficiency of different processors is indeed different. This is a result of the global memory access penalty and the different bus access priorities.

The question of load balancing is further complicated by the issue of granularity. In particular, the system might be prevented from performing any load balancing because the program is partitioned into activities with the wrong granularity. The experiment reported in this section shows how all of this is tied together.

The methodology used is to how much time it would take to perform a constant amount of work using different numbers of activities. The work is 100,000,000 assignments to a global variable. This is divided equally among a certain number of activities, that are then executed on a 10-processor system. The code for this experiment is simply

```
exp = 0;
for (i=1 ; i<100000000 ; i=i*10) {
    exp++;
    start = get_loctime();
    pparfor int j; 1; i; 1;
    {
        int l, k, n;
        l = 100000000/i;
        for (k=0 ; k<l ; k++)
            n = g;
        res[exp][get_pid()]++;
    } epar
    stop = get_loctime();
    time[exp] = stop - start;
}
```

The experiment is divided into phases, each of which does the same work with a different number of activities. The res array counts the number of activities performed on each processor in each

number of activities	*elapsed time [s]*
1	384.350
10	100.230
100	100.210
1000	85.750
10000	85.625
100000	95.185
1000000	170.975
10000000	1154.620

Table 4: *Elapsed time for computations with different numbers of activities.*

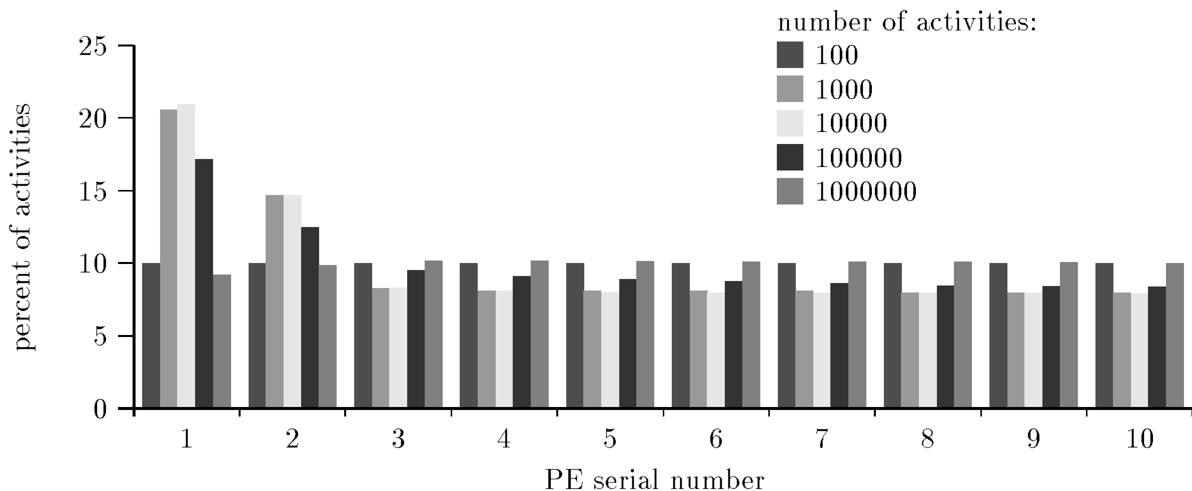

Figure 7: *Number of activities executed on each processor.*

phase. The `time` array tabulates the elapsed time of each phase.

The results of the elapsed time are given in table 4. The best speedup, which is about 4.5, is achieved when between 1000 and 10000 activities are used (a more accurate measurement showed the optimal number to be 1900 activities, which took 85.005 seconds). Using 10000000 activities is three times slower than doing the work serially, because the granularity is much too fine. Most of the time is spent in overhead just creating all those activities.

For each measurement (identified by the number of activities used), the percentage of activities executed on each processor is shown in fig. 7. The results indicate that the best performance is obtained when the loads are not numerically equal. Three cases may be distinguished:

1. The number of activities is too small the their granularity is too large. In this case a more efficient processor may finish executing its activity before less efficient processors, but by the time it finishes all the activities have been picked up already. Therefore it idles, and the time

is determined by the slower processors.

2. The optimal case: a medium number of activities with medium granularity. In this case processor #1 does $2\frac{1}{2}$ times more work than the others. Its increased efficiency is due to the fact that both the spawn descriptor and the global variable that are used in the computation are in its local memory. Processor #2 also does a relatively large amount of work. This is due to the fact that it has the highest bus priority of all the processors that get work through the bus.

3. The number of the activities is too large and their granularity is too small. In this case the contention for the spawn descriptor takes its toll, and limits the performance. All the processors execute about the same number of activities, but it takes them a long time because they have to wait for access to the descriptor. Processor #1 becomes relatively slow, because the heavy burden on its local memory slows it down.

8 Synchronization Overhead

ParC has three synchronization mechanisms: an atomic fetch-and-add instruction (`faa`), a barrier synchronization instruction (`sync`), and semaphores. In this section we measure the performance of the first two, and show how it depends on the number of activities involved.

8.1 Fetch-And-Add

The `faa` instruction is implemented by a set of locks. The address of the variable in question is hashed to one of the locks. The atomic test-and-set instruction from the 386 instruction set, which is supported across the Multibus-II, is used to obtain the lock. Busy waiting is used if the lock is not available. Once the lock is obtained, the `faa` operation is simulated.

Obviously this implementation serializes concurrent `faa` instructions to the same variable. The following code was used to gauge the effect of this serialization:

```
start = get_loctime();
lparfor int i; 1; proc_no; -1;
{
    int *p, j;
    p = &g;
    /* or p = ptr[ (i==proc_no)?1:i+1 ]; */
    for (j=0 ; j<ITER ; j++)
        faa( p, 1 );
} epar
stop = get_loctime();
```

As indicated by the comment, two cases were checked. In the first, all the activities accesses the same variable. In the second, each accessed a distinct variable placed in the memory of another processor.

The results are plotted in fig. 8. As expected, the overhead rises with the number of processors. When the `faa`s are directed at distinct variables, the relationship is linear: it reflects the global

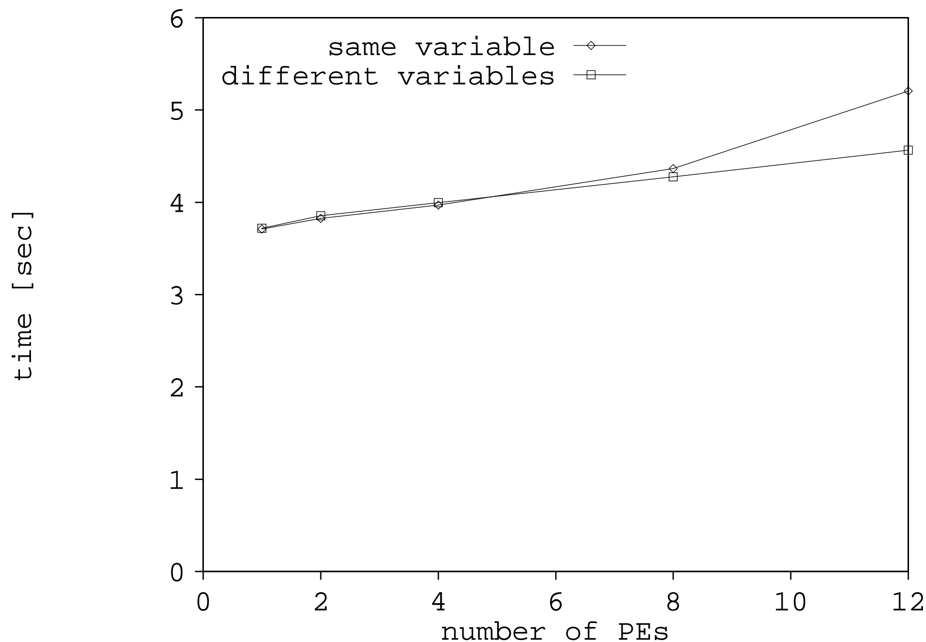

Figure 8: *Time for 10000* `faa` *instructions by each of a set of processors.*

operations on the bus. When all the `faa`s are directed at the same variable, the relationship becomes superlinear for a large number of processors. This is a result of the contention for the same lock. Note that the overhead is quite large relative to the simple nature of the operation (about 0.4 ms per `faa`). The extra overhead is due to raising the priority of the activity when it holds the lock, so as to prevent it from being preempted.

8.2 Barrier Synchronization

The `sync` instruction imposes a barrier synchronization across sibling activities, i.e. across activities that were spawned together by the same construct. The implementation uses a counter and a flag in the spawn descriptor: each activity that reaches the barrier decrements the counter and then spins on the flag. The last to arrive raises the flag and releases all its waiting siblings (it also sets things up for the next barrier). In order to avoid excessive waste if there are more activities than processors, the busy-wait loop includes an instruction to yield the processor if the flag is not up.

The counter is decremented using `faa`, so there is an obvious serialization of this procedure. In addition, when there are more activities than processors, context switches are needed so that all will run and reach the barrier. The experiments in this subsection quantify these two effects.

Experiment 1

In this experiment there is exactly one activity on each processor. These activities synchronize repeatedly, and the time required is measured. The code used is:

```
start = get_loctime();
```

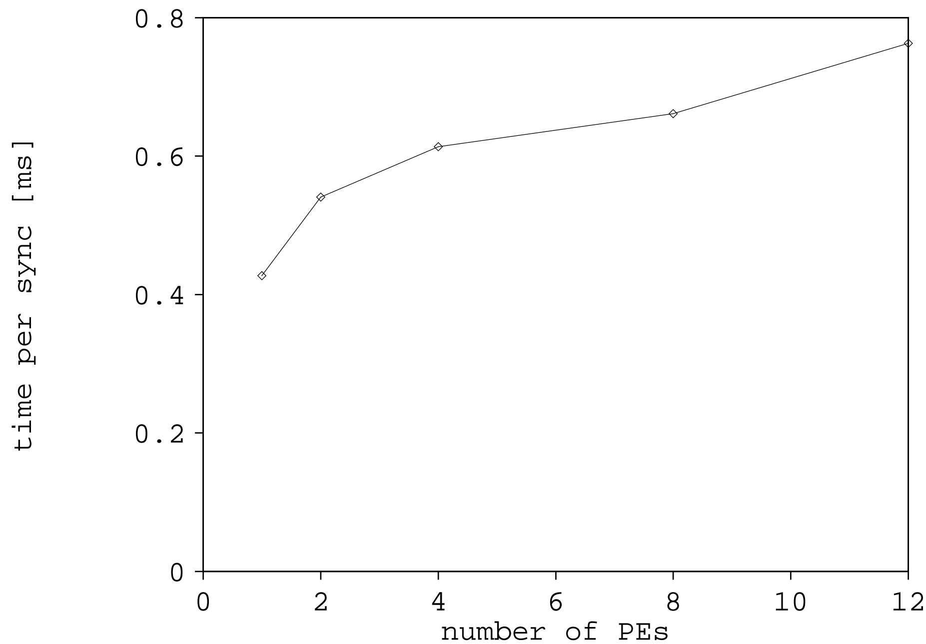

Figure 9: *Time for* sync *instruction by a set of processors. Average over 10000 is shown.*

```
lparfor int i; 1; proc_no; -1;
{
    int j;
    for (j=0 ; j<ITER ; j++)
        sync;
}
epar
stop = get_loctime();
```

The results are shown in fig. 9. The measurement for one processor is much lower than the rest because everything is local. Note that the measured time includes the time for the loop itself. Measurements show this to be 0.0015 ms per iteration, so the difference is small.

Experiment 2

When the number of activities exceeds the number of processors, context switches are needed to allow them all to run and reach the barrier. This is greatly exacerbated if all the activities do not exist yet when the first reach the barrier.

In this experiment 10 processors are used. Three types of measurements are made:

1. A large number of activities are spawned, and they synchronize repeatedly for a large number of times. Thus in effect we can measure the average time to perform a `sync` when all the activities have been spawned already. This is a direct extension of the previous experiment; it is shown by the "just sync" graph in fig. 10. The results indicate a linear relation between

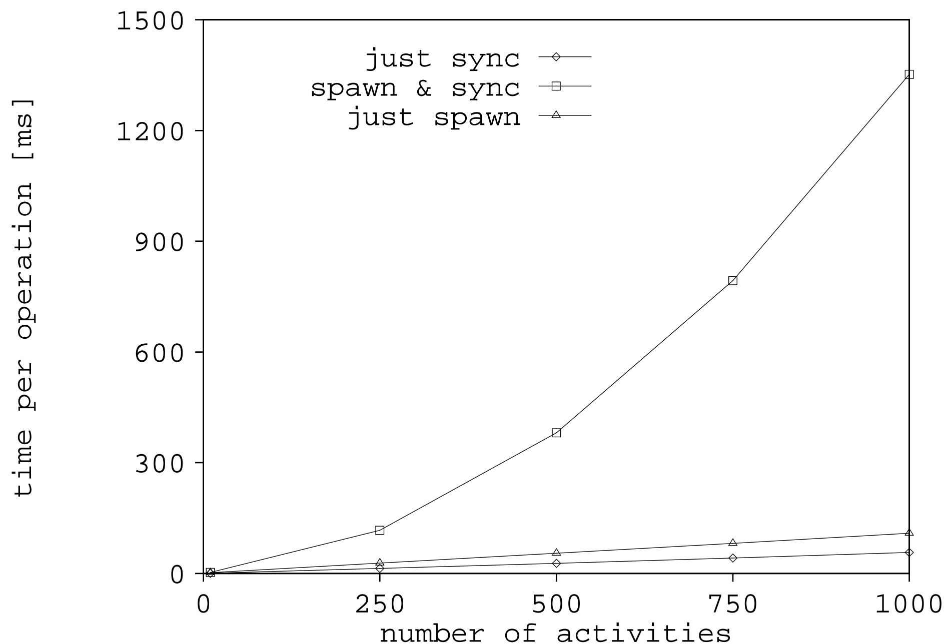

Figure 10: *Time for* `sync` *instruction by a large number of activities. Performance degrades sharply if the activities have to be spawned.*

the number of activities and the time required, which is expected considering that the implementation of the `sync` is actually serial. The constant of proportionality includes the time for the `faa` on the counter and the context switch.

2. A large number of activities are spawned and they synchronize once. Thus the price of spawning also enters the picture. This is shown in the "spawn & sync" graph in fig. 10.

3. To calibrate the previous measurement, we also measured the time just to spawn the tasks ("just spawn" graph in fig. 10). As expected, this is linear in the number of activities, and the constant of proportionality is about 1 ms per activity.

The important effect to notice is that the time to spawn and sync is much larger than the sum of its parts. In fact, it seems to have a quadratic relation to the number of activities. This is a result of the way that the `sync` is implemented, and specifically, a result of using busy waiting with a `yield` instruction. The system then operates according to the following scenario. First, one activity is spawned by the `get_work` task. This activity tries to `sync`, finds that its siblings have not arrived yet, and `yield`s. `get_work` runs again, and spawns another activity. This activity too tries to `sync`, fails, and `yield`s. The first activity is then rescheduled, tries again, fails again, and `yield`s again. This pattern is repeated until all the activities are spawned: after each one, *all* the existing activities try to `sync`. Thus the time to spawn and `sync` N activities is proportional to the sum of N times the spawn overhead, plus $(N/P)^2/2$ times the overhead to perform a context switch and check the sync condition.