USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

BLASTH, a BLAS library for dual SMP computer.

Guignon Thomas
Laboratoire ASCI
Orsay, France
guignon@asci.fr, http://www.asci.fr/

Abstract

This paper presents a multi-threaded BLAS library for dual SMP Intel computer running Linux. We present simple techniques to obtain parallelism for BLAS call transparently from the client program. We discuss some synchronization methods available under Linux, show performances results for a representative set of BLAS and for a high level linear algebra kernel. We then explain some key points on cache management and how they can impact on performances of the blasth library. Next we'll draw some conclusions on the use of SMP computer for linear algebra and present evolution perspectives for the library.

1 Introduction

BLAS[1][8, 4, 3] are a set of subroutines written in Fortran 77 which provide a standard API for simple linear algebra operations. BLAS are now widely used and base of library such as LAPACK, PETSC, SCALAPACK... A recent development in parallel computing was to use on the shelves components to build high performance computers, these components could include low and dual Intel processor systems. This choice leads to the use of two parallel programming models: message passing between nodes end shared memory inside nodes. The goal of the blasth library is to provide a transparent support of shared memory for BLAS call: the program call BLAS as usual but the call is processed on 2 processors. BLAS are split in 3 level:

- level 1: vector operations,
- level 2: matrix-vector operations,
- level 3: matrix-matrix operations and triangular solve.

In this paper we first present some discussion and experiments for synchronizing threads with Linux on SMP computer, we will then focus on daxpy and ddot level 1 BLAS, dgemm (from level 3). The sequential BLAS library used will be the one from the f77 implementation ASCI Red project[2] and ATLAS[3][9] (for level 3). We also present results for a block LU factorization.

[1] from Basic Linear Algebra Subroutines

2 Base principles

Parallelism in BLAS is transparent from the client program: only a call to setup the execution environment and changing the subroutine calling names is needed. The execution scheme is master-slave: the master process runs the program while the slave process is waiting for instruction for master. When the master issue a BLAS call it tells the slave what job to do by communication via shared memory and each one does his "job part". The master waits for the slave to finish his job and continues the normal execution of the program.

The implementation uses The Linux Thread Library which is included with glibc package. This library provides very simple view of shared memory because each thread has the same memory space (data and stack). Note that The Linux Thread Library does not use real threads but traditional Linux processes sharing their memory space so in the following we will use the 2 words thread/process with the same meaning.

2.1 Synchronization

The master-slave approach relies on process synchronization at start and end of BLAS call. Under Linux process synchronization can usually be done in two way:

- IPC semaphores.
- Thread semaphores.

One third way is to use a shared memory variable (synchronization variable): the slave spins on testing the value change of this variable. In this case the slave is always running even if it doesn't make useful work and a normal (observed) behavior of the Linux scheduler is to place the

[2] http://www.cs.utk.edu/~ghenry/distrib/
[3] http://www.netlib.org/atlas/index.html

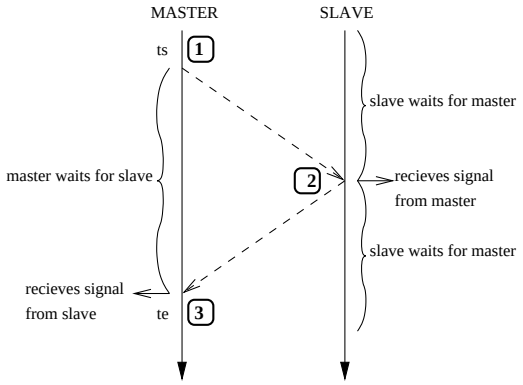

Figure 1: ping-pong test events.

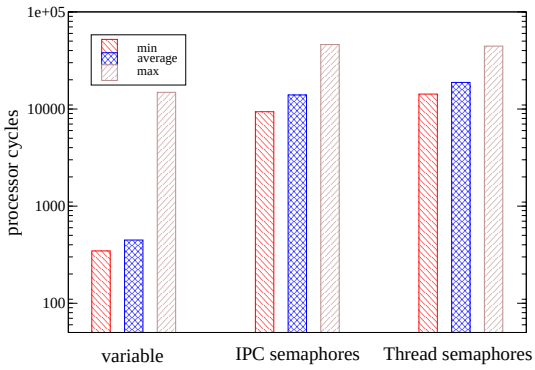

Figure 2: min, average and max times for ping-pong test (1 cycle = 1/400e6 s.).

each running process on different processors so when the master process need synchronization the slave is immediately ready and running.

We compare the three alternatives in a ping-pong test: we make the slave wait for the master and the master "signals" the slave; next the master wait for the slave and the slave "signals" the master . We measure the number of cycles on the master to get the whole job done. Figure 1 may help in understanding the ping-pong test. Moreover each synchronization method must ensure that:

- the slave cross point 2 if and only if the master cross point 1,
- the master will go through point 3 if and only if the slave cross point 2.

The synchronization variable method fulfills the previous requirements. For IPC and Threads semaphores we must use 2 semaphores[4] each one indicates when the master and slave are ready to enter in parallel section; the ping-pong is done with 2 barriers that act like this:

- each semaphore holds value 0,
- master posts on semaphore 0 (semaphore 0 holds 1) and waits on semaphore 1 value becomes 1.
- slave posts on semaphore 1 (semaphore 1 holds 1) and waits on semaphore 0 value become 1.
- each semaphore holds value 0 again.

Experiments are realized on a dual PII 400 and the time measurement is done using the time stamp counter[5] (tsc) of Intel Pentium processors, we suppose that the tsc of each processor holds roughly the same value. Results are presented figure 2: for each method we make 1000 runs and presents the smallest, average and largest time for ping-pong. These results show that synchronization variable is by far the fastest method. As we said previously the difference between synchronization variable and the 2 other methods is that the slave process is ready and running so synchronization does not pay the cost of a system call and moving process from the wait/suspend queue to the running queue. On the other hand having an active slave may interfere with other multi-threaded library.

[4] it can also be done with mutexes.

[5] this is a 64 bits counter which value is incremented each cycle and start counting at power on of processor

2.2 Passing parameters and data sharing

Passing BLAS parameters to the slave is realized by writing in a shared variable the address of the first parameter of the BLAS call before synchronization as shown in the following example `env_base` holds a pointer to the data needed by slave process and `env_blasth_signal_value` holds the function to be run by slave:

```
void **env_base;
void (*env_blasth_signal_value)();

void blasth_daxpy(const int *n,
                  double *alpha,
                  double *X,
                  const int *incx,
                  double *Y,
                  const int *incy){
 // realize Y = *alpha * X + Y
 // where X and Y are vectors of
 // size *n with respective increments
 // of *incx and *incy
 // executed by the
 // master from the
 // application program

 env_base = (void **)&n;
 env_blasth_signal_value = TH_DAXPY;

 // tell the slave there is
 // some job to do
 blasth_master_sync();
```

```
 // some job

 // wait for the slave
 blasth_master_sync_end();
}


void blasth(){
 // excuted by the slave from the
 // environement setup

 while(1){
  // wait for the master
  blasth_sync();

  //call the function set by the master
  env_blasth_signal_value();
 }
}

TH_DAXPY(){
 // at this point env_base
 // contains a pointer to the
 // first needed parameter
 // (int *)env_base[0] is a
 // pointer to the size of vectors (*n)
 // (double *)env_base[1] is a
 // pointer to the scaling factor (*alpha)
 // (double *)env_base[2] is a
 // pointer to the first element
 // of vector X
 // ....

 // some job

 // tell the master
 // that job is finished
 blasth_sync_end();
}
```

The blasth_daxpy calling sequence is identical to the daxpy calling sequence from a C program (the BLAS library is originally written in f77 so the API is f77 compliant) and the parameters are written before the synchronization variable so the strong memory ordering (for write operations) of the Pentium processor family ensure that slave process will see exactly the same parameters in TH_DAXPY as the master in blasth_daxpy.

Data sharing is done by splitting the result between the master and the slave: if the result is a vector the master has to construct the first half and slave has to construct the second half; if the result is a matrix of size $m \times n$ the master will construct either the first $n/2$ columns or $m/2$ rows and the slave will construct the remaining columns or rows. We show splitting examples in figure 3 for dgemv and dgemm (respectively matrix vector product and matrix matrix product).

We does not use cycling split of data to avoid cache line sharing between processors (especially when writing data). The splitting are also chosen to avoid temporary data which would require dynamic allocation.

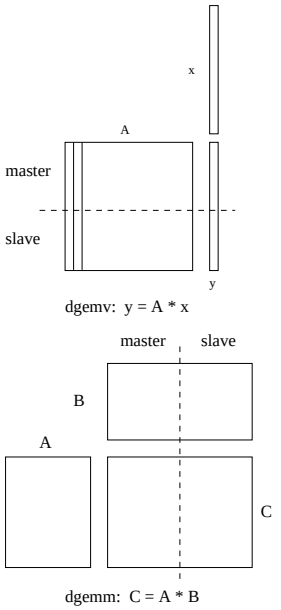

Figure 3: data splitting for dgemv and dgemm

3 Some results

The results presented here where done on a dual PII 400 system running Linux The time measurements are done using the time stamp counter of the processors. The base BLAS libraries used are from the Fortran 77 implementation[6], ASCI Red project and ATLAS project (for level 3). The memory bandwidth measurements are done with the hardware performances counter available on the Pentium Pro family processors[7]. For each BLAS we test we compare single and dual performances in two case, with cache memory flushed out (datas are read from main memory) and with cache memory loaded: the test is done several times before measuring execution time (remark: this does not seem that data will fit into cache).

3.1 daxpy and ddot

daxpy and ddot are the two main level 1 BLAS, daxpy is a linear combination of vectors $y = \alpha.x + y$ and ddot is a dot product of the form $\alpha = x^T.y$ (the "d" before each name indicates that the BLAS use double precision floating point arithmetics).

[6] with recent version f77 level 1 blas compete with those of ASCI Red project

[7] an introduction on how to use these counters is available at http://www.cs.utk.edu/ ghenry/distrib/mon_counters

Performances results for daxpy are presented figure 4 and acceleration is showed 5. Observed results are typical for level 1 operations: we observe a peak when data fits into L1 cache and a smooth decrease when data does not fits in L1 but remains in L2. Performance from main memory is driven by memory bandwidth. Acceleration is quite good for large vectors especially when datas are in cache.

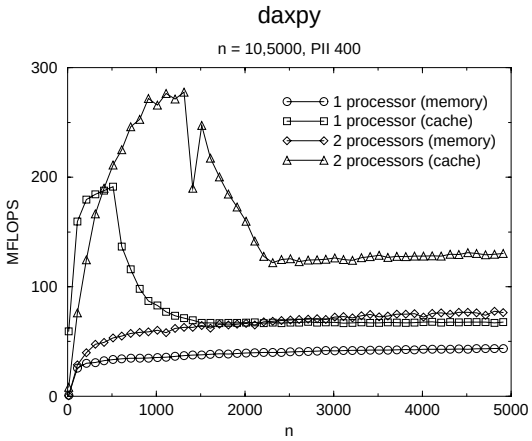

Figure 4: daxpy performances with 1 and 2 processors

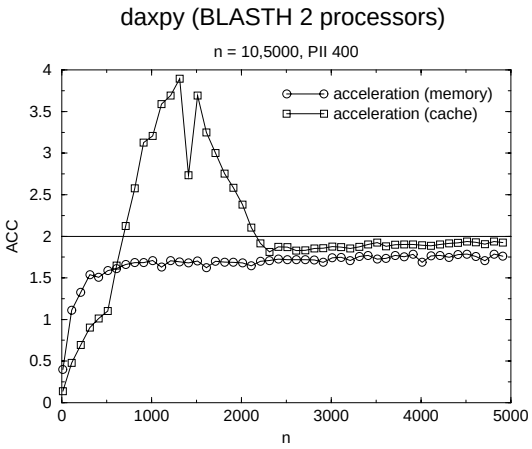

Figure 5: daxpy acceleration for 2 processors

Performance results for ddot are showed figure 6 and figure 7 for acceleration. Comments on performances results are the same as for daxpy but we observed that scaling is not as good as for daxpy: cache operation scaling is roughly 1.8 but remains good. On the other side memory operation scaling in poor (≤ 1.5). To understand that we make some memory bandwidth measurements with very large vectors to not consider time spent in synchronization and the results are presented in table 1: we see that single processor ddot use more than half of the theoretical peak memory bandwidth (the system uses pc100 SDRAM allowing 800e6 B/s memory bandwidth) and dual processor uses up to 75% of available bandwidth which seems very good for the test system.

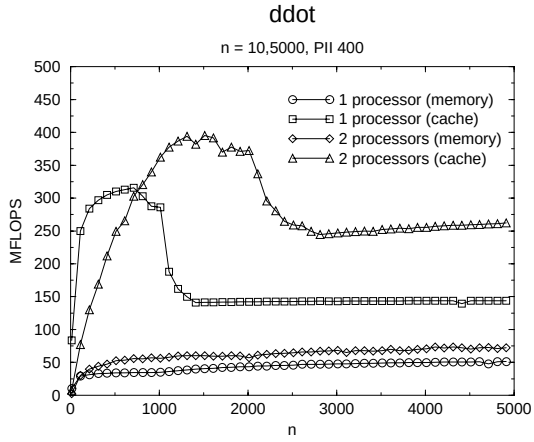

Figure 6: ddot performances with 1 and 2 processors.

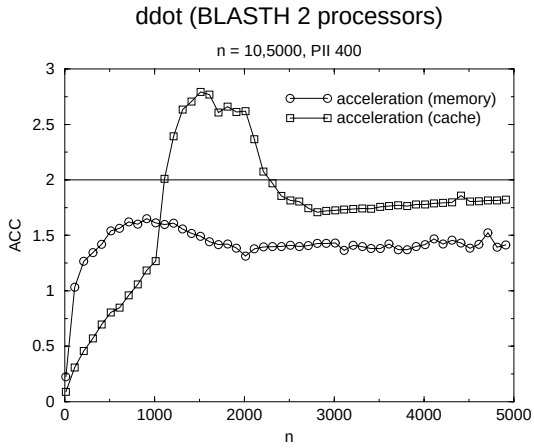

Figure 7: ddot acceleration for 2 processors.

The daxpy and ddot BLAS show good acceleration with the blasth library when datas are present into cache memory, this behavior is in fact common to all level 1 operations since each component of vectors is used only one time in computation (no temporal locality). Operations on small vectors ($n \leq 1000$) will not scale due synchronization cost compared to the small number of floating point operations issued (n or $2.n$ for level 1 BLAS). The memory bandwidth is the key point for scalability when datas are out of cache which is always true for very large data sets.

3.2 dgemm and block LU factorization

dgemm is a level 3 operation which performs a matrix matrix product $C = \alpha.A.B + \beta.C$ where A, B, C are respec-

nproc	n	memory bandwidth (1e6 B/s)
1	10000	440
1	100000	470
2	10000	590
2	100000	600

Table 1: memory bandwidth used by ddot with 1 and 2 processors.

tively $m \times k$, $k \times n$ and $m \times n$ matrices. The ASCI Red implementation and ATLAS use a block method to perform matrix matrix multiply and achieve good performances As we see in figure 8 end figure 9 performances and acceleration are very good and remain as the size of matrices increase. Performances for dgemm are important because it's a building block for block LU factorization. We know outline a block LU factorization method, the reader may refer to [5] for an in-deep analysis and algorithms, and discuss how multi-threaded version can be realized.

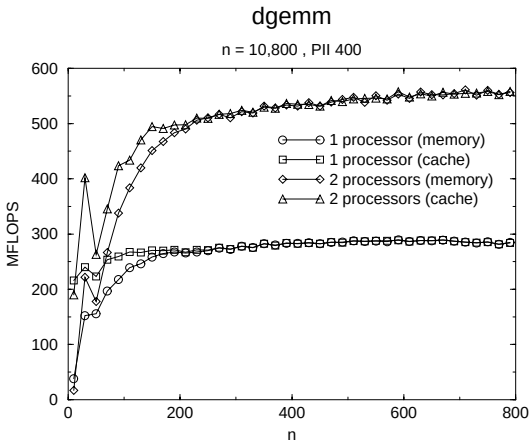

Figure 8: dgemm performances with 1 and 2 processors

LU factorization is used to solve linear system like $A.X = B$ by factorize A into $L.U$ where L is a lower triangular unit matrix and U is an upper triangular matrix. Efficient LU methods rely on matrix matrix product and we outline such method in the following. For simplicity we suppose that the block size n_b divide n which is the size of the square matrix A and set $N = n/n_b$. Each block of A is numbered $A_{i,j}$ $i, j = 1..N$. The block LU performs as follow:

1. set $k = 1$,
2. compute $A_{k,k} = L.U$ and overwrite $A_{k,k}$ with L and U using a non blocked LU factorization,
3. apply row interchange in $A_{k,k+1:N}$ and in $A_{k,1:k-1}$
4. solve $L.X = A_{k,k+1:N}$ and overwrite $A_{k,k+1:N}$ with the solution,

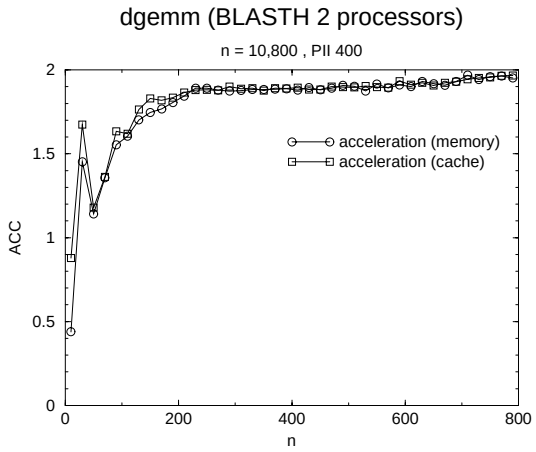

Figure 9: dgemm acceleration for 2 processors

5. solve $X.U = A_{k+1:N,k}$ and overwrite $A_{k+1:N,k}$ with the solution,
6. update $A_{k+1:N,k+1:N}$ with $A_{k+1:N,k+1:N} = A_{k+1:N,k+1:N} - A_{k+1:N,k}.A_{k,k+1:N}$,
7. if $k < N$ set $k = k + 1$ and go to step 2.

Step 2 uses subroutine dgetf2 from LAPACK, row interchange is done using dlaswp, step 3 and 4 use dtrsm (triangular solve with multiple right hand side) and step 5 uses dgemm. It is important to note that most computation is done in steps 4 and 5 which use level 3 BLAS and represent $1 - 1/N^2$ of the floating point operations issued (see[5]). Figure 10 presents task dependencies in block LU for step k, it outlines a graph dependency for a parallel implementation. At this point we have two choices for multi-threaded version:

- We can use a dependency graph of task with each processors searching for ready tasks (tasks for which each ancestor is done) and executing them. By splitting the solve and matrix multiply tasks into smaller ones we can see that LU for step k can be done while matrix multiply for step $k - 1$ is finishing since $A_{k,k} = A_{k,k} - A_{k,k-1}.A_{k-1,k}$ is done at step $k - 1$.
- We can also perform the LU task on one processor and then use multi-threaded version of dlaswp, dtrsm and dgemm.

The first solution requires to construct the dependency graph and more complex synchronization scheme but always based on synchronization variable. Construct the graph and searching for ready task may be a serious overhead thus limiting acceleration. The second solution let the LU task be a sequential bottleneck and use parallelism only on level 3 operations.

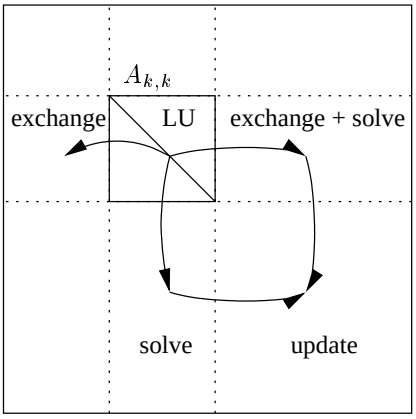

Figure 10: LU factorization scheme for the step k

For implementation we choose the second solution because it's simple and LU bottleneck is in fact very small since it represents only $1/N^2$ of the overall job. Results for our block LU factorization are presented figure 11 and figure 12: we use dgetrf from ATLAS as a reference code for our sequential block LU. Acceleration figure presents two curves; one presents acceleration obtained by the multi-threaded block LU, the other (ideal acceleration) is computed using the sequential code by looking for time spent in dgetf2 and in level 3 operation:

$$acc_{ideal} = \frac{1}{t_{dgetf2} + t_{level3}/2}$$

For $n \geq 1000$ acceleration is relatively closed to ideal acceleration but smaller case acceleration is far from ideal due to synchronization cost.

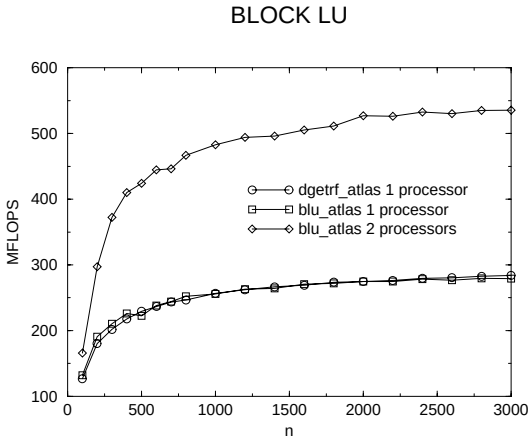

Figure 11: block LU performances with 1 and 2 processors

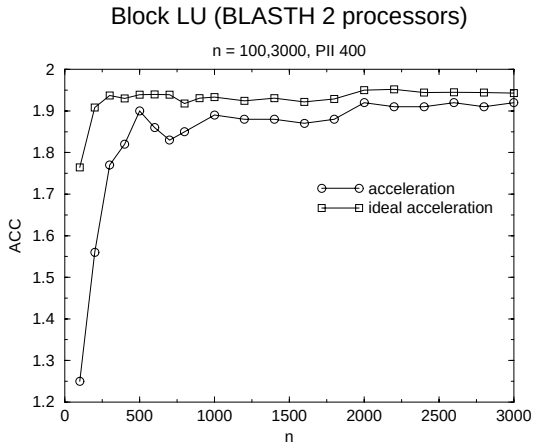

Figure 12: block LU acceleration with 1 and 2 processors

4 Memory bandwidth and cache use

In this section we present some well known key points of cache use and how they impact on performances in BLAS subroutines for single and dual cpu execution, we will discuss on effect of non continuous data, false sharing, mutual exclude, data blocking, stack alignment and thread/processor affinity. All examples suppose we are using an Intel P6 class processor which suppose that L1 caches lines are 32 bytes long and L1 is 2 way set associative, reader may refer to[1] for full information on optimizing codes for Pentium processors.

continuous datas.

Level 1 BLAS use loops that access arrays in a sequential manner; if we suppose that an array t[n] of double is cache line aligned (for simplicity) accessing t[0] loads t[0],t[1],t[2],t[3] into one level 1 cache line, then following array cell accesses may not use memory until we access t[4]. This situation makes the ration of useful loads[8] on effective loads[9] be 1. By using vector increments of 2, only even cells are used which make t real size becomes 2n, and the previous ration becomes 0.5. This is a first argument to avoid cycling split of vectors for multi-threaded level 1 BLAS because master and slave processes will use more memory access to do the same job.

false sharing.

The Intel P6 family use a cache coherency protocol with four states usually called MESI[10]. This protocol is write invalidate which means that when two or more processors holds a copy of the same memory line if one of them writes

[8] loads necessary to perform computation
[9] loads effectively issued
[10] Modified, Exclusive, Shared and Invalid

in, the cache line holding the memory line is invalidated on other processors. False sharing occur when processors write to a shared cache line but not at the same location: there is no real coherency problem since processors write to different location and since the cache allocate a line when a write misses[11] the protocol makes each processor invalidate the other forcing reload of a cache line at each write. This situation occurs with blasth library at the boundary of results blocks but in the case of level 1 BLAS only one cache line will be shared between 2 processors. In the case of dgemm for a $m \times n$ matrix up to $max(m, n)$ caches lines can be shared but usual optimization of dgemm use block copy of the resulting matrix avoiding such situation. False sharing is another argument to avoid cycling split and we can see effect on daxpy in figure 13: there is no cache effect on operand y while operand x is accessed by each thread with an increment of 2.

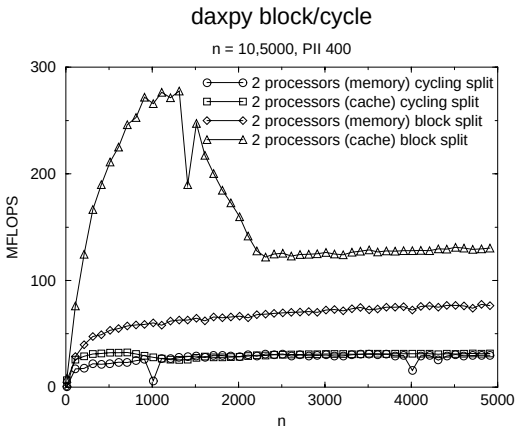

Figure 13: Effects of false sharing with daxpy on 2 processors

mutual exclusion.

Mutual exclusion appends when 2 or more memory lines are needed but cannot be in cache all-together because they fit in the same cache line and successive access cause exclusion of the memory lines previously loaded. This is a real problem for direct mapped caches where a memory line can be in only one cache line. N way set associative caches solve this problem by allowing a memory line to be in N different cache lines, the N locations are called a set; this is the case of the P6 processors where L1 cache is 2 way set associative and it avoids mutual exclusion for all Level 1 BLAS with less than 3 vector operands (like ddot and daxpy). Mutual exclusion can also appends for matrix operations like dgemm: when using a block method, leading dimension[12] can be such that some memory lines share the same set into cache avoiding more than N of them at the same time: on figure 14 the memory lines l1 l2 and l3 are 1024 doubles spaced (remember that PII L1 cache has 256 sets each containing 2 line of 32 bytes) so they fit into the same L1 cache set which can hold only 2 different memory lines. Thus l1, l2 and l3 and subsequent lines exclude mutually. The solution to have the block in cache is to make a copy into contiguous memory and this is the solution adopted in ATLAS.

[11] PII and up use a write-back/write-allocate strategy while the PPRO use write-trough.

[12] distance between first elements of columns.

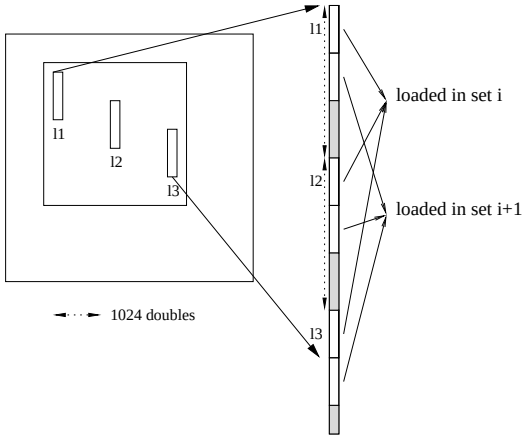

Figure 14: Mutual exclusion in block access.

data blocking.

Blocking is an optimization technique that allows a full cache use and thus reduces memory bandwidth usage. Blocking is no always possible: for BLAS 1 and 2 the majority of data is accessed only one time making temporal locality very low; on the other hand Level 3 operations use a three nested loop structure with 2 dimension matrices making each matrix elements accessed more than on time. We give an example for matrix matrix product $C = A.B$ where C, A and B are respectively $m \times n$, $m \times p$ and $p \times n$ matrices; in this case each element of A and B are accessed respectively n and m times. The block method for matrix matrix product generally consist of:

- split result matrix C into blocks $C_{i,j}$ of size $n_b \times n_b$, each blocks is constructed into a continuous array C_b which is then copied back into the right $C_{i,j}$.
- matrices A and B are split into panels A_i and B_j of size $n_b \times m$ and $k \times n_b$ each panel is copied into continuous arrays A_b and B_b. The choice of n_b must ensure that C_b A_b and B_b fit into one level of cache, usually L2 cache.

then:

1: **for** $i = 1$ to m/n_b **do**
2: $A_b \leftarrow A_i$

3: **for** $j = 1$ to n/n_b **do**
4: $B_b \leftarrow B_j$
5: $C_b \leftarrow 0$,
6: **for** $k = 1$ to p/n_b **do**
7: $C_b \leftarrow C_b + A_{bk}.B_{bk}$,
8: **end for**
9: $C_{i,j} \leftarrow C_b$
10: **end for**
11: **end for**

We suppose for simplicity that n_b divides m, n and p. Figure 15 may help in understanding operations performed on blocks. In the case of the previous algorithm matrix A is loaded only one time into cache compared to the n times access of a classical ijk loop while matrix B is still accessed m times. This simple block method greatly reduce memory access and real codes may choose by looking at matrix size which loop structure (ijk vs. jik) is best appropriate and if some matrix operand fits totally into cache.

In the previous we where working on L2 cache and we does no talk about L1 cache use. In fact L1 will be generally too small to handle a $C_{i,j}$ block and one panel of A and B but remember that operation performed at step 7 of the previous algorithm is a matrix matrix product so each operand A_{bk} and B_{bk} is accessed n_b times: this part could also use a block method. Since n_b is relatively small the implementation may load only one of C_b, A_{bk}, B_{bk} into L1 cache and works with others from L2 cache. The reader may refer to ATLAS source code and description for a complete analysis and test of block methods in various environments. Another projects for fast matrix matrix multiply are Phipac[13][6] and the BLAIS[7] library from MTL[14].

stack alignment and thread/processor affinity.

Stack alignment has been an issue because older gcc versions cause doubles not being aligned on an 8 bytes boundary which make access cost extra cycles. We have face this problem with level 3 BLAS that use fixed size arrays on stack for blocking resulting in poor performances. Recent gcc versions (i.e. 2.95.x) solve this problem and propose various option to control the stack alignment such as -malign-double and -mpreferred-stack-boundary=x.

Thread/processor affinity is a general issue in smp systems; the cache efficiency can be reduce if the task scheduler moves thread from one processor to another. At this time there is no way to force thread/processor affinity on a standard Linux kernel but as we said in section 2.1 the normal behavior of the Linux scheduler is to place each running process (master and slave) on 2 different processors and a kernel patch is available at http://isunix.it.ilstu.edu/∼thockin/pset/ that add some control on the thread/processor binding.

[13] http://www.icsi.berkeley.edu/∼bilmes/phipac/
[14] http://www.lsc.nd.edu/research/mtl/

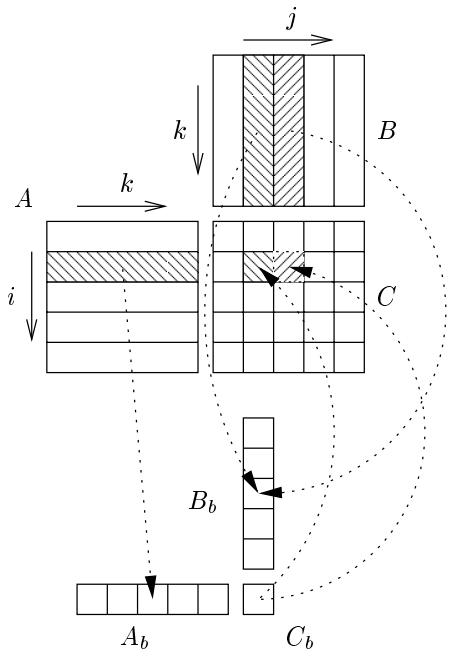

Figure 15: Block matrix matrix product.

5 Conclusions and perspectives

The use of low end Intel SMP computers inside Beowulf or c.o.w. can help in getting better performances when applications does not consume a lot of memory bandwidth: we have always to remember that a cluster of single processors nodes has twice aggregate memory bandwidth of an equivalent cluster of dual SMP with the same number of processors. In linear Algebra a lots of high level computation kernel use block methods and the blasth library could help in these cases as we see for LU factorization. The perspectives for this work are important because we need to work on more LAPACK routines to provide a useful library. There is also work to do on matrix matrix product and LU factorization to improve acceleration in small cases and our choice of simple parallelization for LU cannot be ideal for more processors. Porting to other platforms such as alpha systems is an ongoing work and theses systems can change scalability results for high bandwidth consuming BLAS such as ddot and dgemv making the blasth library more interesting.

References

[1] Intel Architecture Optimization Manual. Order Number 242816-003, 1997.

[2] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Corrigenda: “An extended set of FORTRAN Basic Linear Algebra Subpro-

grams". *ACM Transactions on Mathematical Software*, 14(4):399–399, December 1988. See [4].

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988. See also [2].

[5] Ch. F. Van Loan G. H. Golub. *Matrix computations*. The Johns Hopkins University Press, third edition, 1996.

[6] C.W. Chin J. Bilmes, K. Asanovic and J. Demmel. The PHiPAC matrix-multiply distribution. Technical Report TR-98-35, International Computer Science Institute, Brekeley CA, 94704, October 1998.

[7] Andrew Lumsdaine Jeremy G. Siek. A rational approach to portable high performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *2th European Conference on Object-Oriented Programming, workshop on Parallel Object-Oriented Scientific Computing (POOSC'98)*, Brussels, Belgium, july 1998.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979.

[9] Jack J. Dongarra R. Clint Whaley. Automatically Tuned Linear Algebra Software. In *SuperComputing '98 Proceedings*, 1998.