Load Balancing Strategies to Implement Non-Uniform Data-Parallel Computations

Salvatore Orlando
Ca' Foscari University of Venice

(joint work with
Raffaele Perego – CNUCE C.N.R. - Pisa)

14 February 1997

ABSTRACT

In this talk we describe a run-time support (SUPPLE) for *non-uniform parallel loops*. The support exploits *dynamic scheduling of iterations* and *dynamic data transfer* of chunks of data. It is completely *asynchronous* and employs *prefetching* to overlap computation with communication.

We report measures obtained by using SUPPLE to implement a kernel application for flame simulation. Experiments have been conducted on a 64-node Cray T3D, on which we compared our results with those obtained running the same application written in CRATF-Fortran (a HPF-like language).

OUTLINE

- Current run-time supports to implement parallel loops on DM-MIMD machines
- Non–uniform parallel loop issues and a sketch of our solution
- The benchmark
- The synthetic input data set
- Our support: SUPPLE
- Experimental results

PARALLEL LOOP RUN-TIME SUPPORTS

State of the art

- *static data distribution*, mainly to exploit *locality*
  - `BLOCKING` data distribution
    - *locality* exploitation
  - `CYCLIC` data distribution
    - to be used if locality is less important than *load balancing*
- *static scheduling* to reduce run-time overheads
  - derived from the data distribution
  - owner-computes rule

NON-UNIFORM LOOPS

- Loops in which the computational load of each iteration may vary significantly

- An example:

```
do k=1 to N_ITER
   forall (i,j=1 to N-1)
      A[i,j] := f(A[i,j], A[i-1,j], A[i+1,j]);
   end forall
end do
```

The execution cost of `f()` is *not constant*
Most of the total load is concentrated in some regions of the array

HPF solutions

- If the array `A[]` is distributed `CYCLIC`
  - we may solve the load imbalance issues, but we may break locality

- If we adopt a `BLOCKING` distribution
  - we reduce communications and exploit locality, but we will pay for the load imbalance

- Combination of `CYCLIC` and `BLOCKING` can be employed, but the best choice depends on the specific problem
  - it is difficult to find an optimal solution at compile time
  - when iterations are scheduled statically, the two distributions becomes solutions for two contrasting goals

SUPPLE – An overview

- Support for unbalanced parallel loops with regular and predictable communication (*stencil*)
- BLOCKING array distribution to exploit locality
- an initial static scheduling strategy based on the owner-computes rule
- *dynamic scheduling* occurring **only if** a load imbalance occurs
  - receiver initiated load balancing technique
  - round-robin search of loaded partners
  - prefetching of requests for further work to overlap communications with computation

THE BENCHMARK APPLICATION

The benchmark we used is directly derived from a class of actual scientific codes used to carry out detailed flame simulations

Saltz et al. Runtime Support and Dynamic Load Balancing Strategies for Structured Adaptive Applications. In *Proc. of the 1995 SIAM Conf on Parallel Processing for Scientific Computing*, Feb. 1995.

Fox, Saltz et al. Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions. *IEEE Trans. on Parallel and Distr. Systems*, 6(8):815–831, Aug. 1995.

Highly detailed simulations give important information on the fuel flammability limits and on the interactions between chemical and physical processes, and its computational requirements for heavier hydrocarbons imposes the exploitation of massive parallelism.

SIMPLIFIED HPF-like TWO-DIMENSIONAL CODE

```
   DO k= 1,K

C Convection phase over a structured mesh
C Constant computation time - Neighboring communications
C 1/4 of the total execution time

        FORALL (i= 2:N1-1,  j= 2:N2-1)
          A[i,j] = A[i,j] + F(B[i,j], B[i-1,j], B[i,j-1], B[i+1,j], B[i,j+1], C[i,j])
        END FORALL

        B = A


C Chemical reaction phase - No communications - High load imbalance
C Workload distribution evolves during simulation
C 3/4 of the total execution time (usually more than half of the total time)

        FORALL (i= 1:N1,  j= 1:N2)
          C[i,j] = Reaction(A[i,j])
        END FORALL

   END DO
```

THE SYNTHETIC INPUT DATA SET (1)

In our *load imbalance model* we assume that there exists a region of the array on which the load is greater than the average

- d is fraction of the array corresponding to this loaded region
- d, t > d, is the fraction of the total load concentrated on this region

where $0 \leq d,\ t \leq 1$

THE SYNTHETIC INPUT DATA SET (2)

An array mapped BLOCKING on 64 procs

This is an array modelling the case $d = 0.1$

A larger/smaller value of t corresponds to a darker/lighter top-left area

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

THE SYNTHETIC INPUT DATA SET (3)

The imbalance, in our model, is

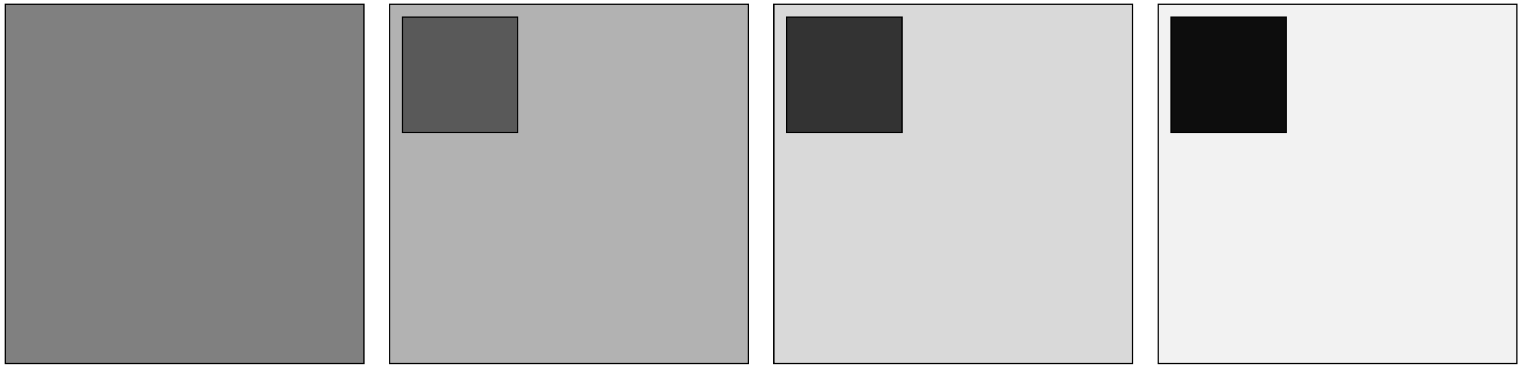
- directly proportional to t
- inversely proportional to d

Factor of imbalance: $\boxed{F = \frac{t}{d}}$

THE SYNTHETIC INPUT DATA SET (4)

Benchmark input data set built according to our model

- The computational cost of the function **Reaction( )** is directly proportional to the value of **A[i,j]**
- For each factor of imbalance F we build a distinct **A[ ]**



F=1 F=3 F=6 F=9

SUPPLE: Load balancing (1)

This part of our support has been used to implement the second *Chemical Reaction* parallel loop:

- BLOCKING data distribution

- Initial static scheduling according to *owner computes rule*

- Iterations scheduled in chunks:
  - each chunk corresponds to a sub-block of the assigned data partition: Tiling of the block
  - the boundaries of each tile are precomputed
  - all the chunks are logically assigned to a local queue LQ
  - a chunk is the minimal unit of scheduling
  - its granularity influences the time elapsed between consecutive polling to probe msg arrivals

SUPPLE: Load balancing (2)

- Work migration only if a *load imbalance* occurs
  - Work migration requires data transfer
  - Remote chunks received are stored in a remote queue RQ
  - Data maintained coherent on the owner, i.e. results of remote executions must come back
  - Coherence messages overlapped with computation
  - Non-blocking handling of data coherence (*full/empty bit* technique)

SUPPLE: Load balancing (3)

- Processor load evaluation
  - each processor measures the execution time of its LQ chunks, and computes its own *average chunk cost*. This cost is communicated when chunks are migrated
  - The processor load is thus evaluated considering the number of chunks in LQ and RQ and the corresponding average execution costs

- Task migration strategy
  - *Receiver initiated* policy for load balancing
  - when a processor understands that its load is becoming low, then it asks a partner for chunk migration: Chunk Prefetching
  - *round-robin* choice within the non-terminated partners
  - a partner broadcasts a *termination* message when its LQ is becoming empty

SUPPLE: Load balancing (4)

- Task selection strategy
  - *Tradeoff* between sending a single chunk (*too much overhead*) and vectorizing a bunch of chunks in single a message (*poor load balance*)
  - we used a modified version of *Factoring*. At the beginning several chunks may be sent together, but this amount decreases when LQ is going to become empy:

$$\frac{N\_CHKS_{unsched}}{2 \cdot P}$$

SUPPLE: Load balancing (5)

- Load balancing profitability determination
  - based on a statically fixed THRESHOLD
  - If the local load of a processor *P* becomes less than THRESHOLD, then *P* asks one of the non–terminated processors (round-robin) for *prefetching* remote chunks
  - The asked processor grants the request only if its local load is greater than THRESHOLD
  - THRESHOLD is also used for termination handling:
    when a processor discovers that its local load is less than THRESHOLD, it broadcasts a termination message

SUPPLE: SPMD code for load balancing (6)

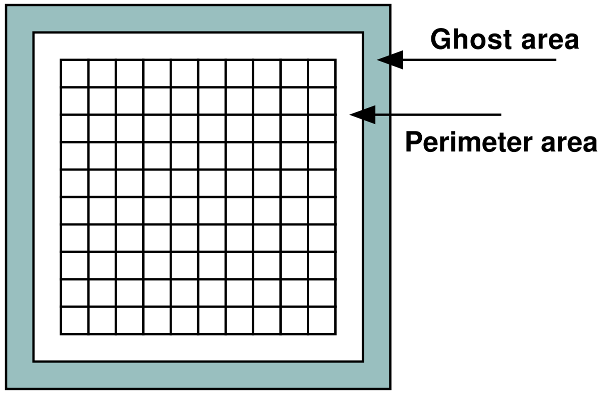
```
Loab_Balancer (queue LQ)
RQ: queue;
T: chunk;
begin
  Initialize_Data_Structures ();
  while (not Terminated&Empty_queues (LQ, QR, partners)) do
    if (My_Load (LQ, RQ) < Threshold) then
      Prefetch_Chunk ( );
      if (* My_Load (LQ) < Threshold for the first time *) then
        Comm_termination ( );
      endif
    endif
    T = Extract(LQ);
    if (Empty (T)) then
      T = Extract (RQ);
      Execute (T);
      Send_Results_Coherence (T);
    else
      Execute (T);
    endif
    Request_Handler ();
  endwhile
end
```

SUPPLE: Stencil communications (1)

This part of our support has been used to implement the first *Convection* parallel loop:

- It adopts standard techniques for *optimizing stencil communications* and *overlapping communication with computation*
- A *ghost area* surrounding the assigned block is allocated on each processor
  - this area is used to receive data from neighboring procs
- The *localized iteration scheduling* are precomputed
  - a *perimeter scheduling*, which corresponds to the iterations assigning the elements on the perimeter of the assigned block
  - an *inner block scheduling*, which corresponds to the remaining iterations
- *ghost* and *perimeter* area have the same **width**. This width is an input parameter of our run-time support

SUPPLE: Stencil communications (2)



- The *inner block* scheduling takes into account the tiling needed to implement the load balancing support
  - the *inner block scheduling* executes chunks of iterations
  - before executing a chunk, it is needed to check the corresponding *full/empty bit*, since we may have to wait for a *coherence msg* if the chunk has been previously migrated

SUPPLE: Stencil communications (3)

The phases of computation are thus the following:

- **sending phase** of the *perimeter area* to the neighboring procs

- *inner block scheduling* of chunks of iterations

- **receiving phase** of the *ghost area* from the neighboring procs

- *perimeter scheduling* of iterations

EXPERIMENTAL PARAMETERS(1)

Some experimental parameters:

- 64 T3D processors
- $1024 \cdot 1024$ arrays of `double`, each one corresponding to a distinct F
- $\mu = 0.038, 0.075, 0.150, 0.300\ msec$, where μ is the average execution time for each point of the *Reaction phase*
- The execution time for each point of the *Convection phase* is $\frac{\mu}{3}$
- The two loops are executed for 10 times, so that the optimal execution time should be: $10 \cdot \frac{4\mu}{3} \cdot (1024 \cdot 1024)$

EXPERIMENTAL PARAMETERS(2)

Some experimental parameters:

- THRESHOLD = from 2 to 40 $msec$, depending on the average execution time
- Each chunk is a tile of 21 doubles $(7 \cdot 3)$
- Some examples of computational load per point
  - $\mu = 0.75\ msec$ and F = 9:
    an unloaded Reaction point is computed in 0.0084 msec, while a **loaded Reaction point** in **0.67 msec**
  - $\mu = 0.75\ msec$ and F = 5
    an unloaded Reaction point is computed in 0.041 msec, while a **loaded Reaction point** in **0.37 msec**

EXPERIMENTAL PARAMETERS(3)

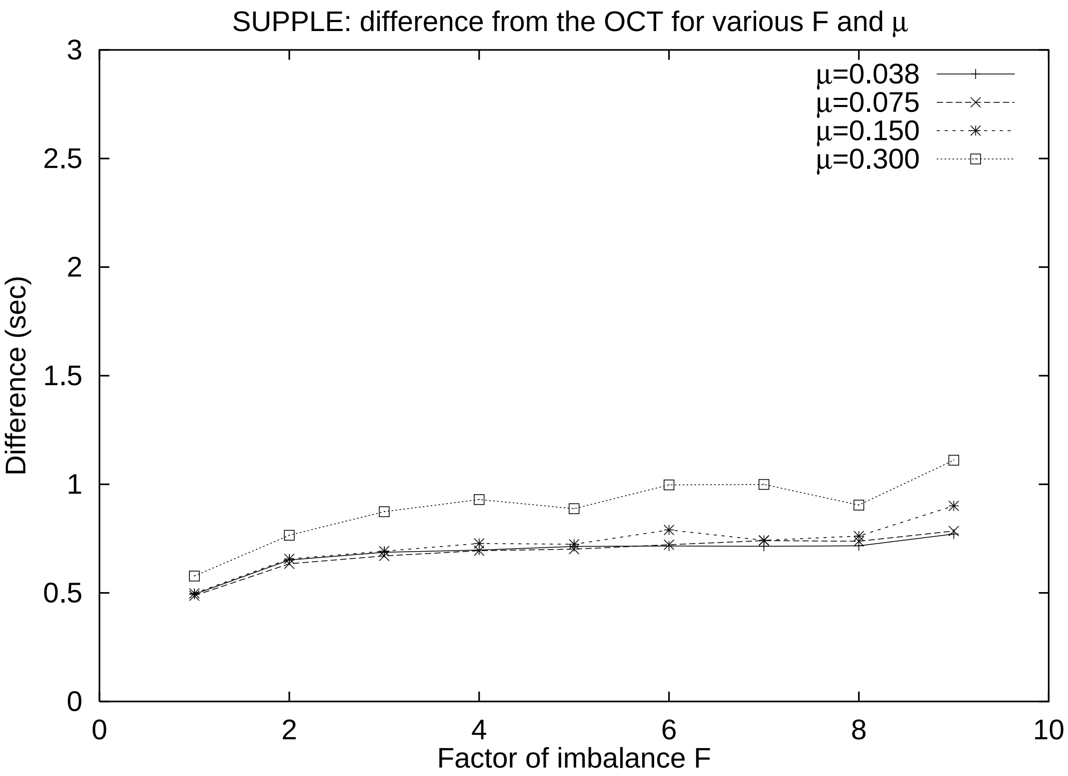
Some heuristics used in the load balancing run-time support:

- A maximum number of ongoing prefetching messages: 2
- When a processor begins to ask chunks, it increases its THRESHOLD value (*10)
- To avoid wrong chunk movements at the beginning due to an incorrect *average local chunk cost*, each processor starts checking the value of its *local load* only when it has already computed several (20) local chunks

SUPPLE: RESULTS

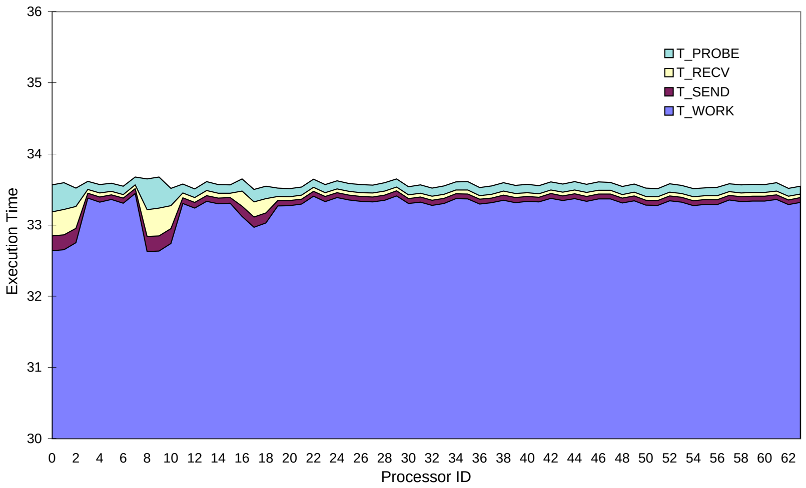
Differences from the Optimal Completion Time (OCT) for several factors F. Each curve is related to a distinct μ.

μ	OCT
0.300 msec	65.5360 sec
0.150 msec	32.7680 sec
0.075 msec	16.3840 sec
0.038 msec	8.1920 sec



SUPPLE: OVERHEADS

- Case $\mathsf{F} = 8$ and $\mu = 0.150\ msec$
- The following graph shows, for each processor:
  - Overheads of MPI routines: send, receive, and probe
  - All the other time spent in useful work and other overheads

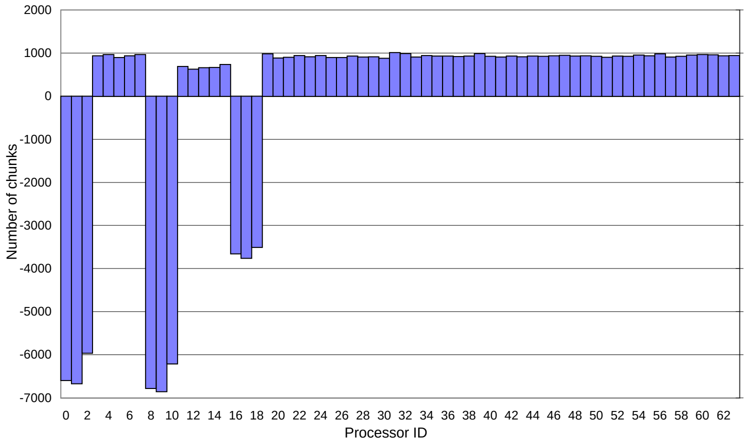


SUPPLE: CHUNK MIGRATION

- Case F $= 8$ and $\mu = 0.150\ msec$. The graph plots, per each processor:
  - The expression (Chks_executed - Chunk_assigned)
  - If no chunks are migrated, the value of the above exp is 0. It is negative if a processor gaves away chunks, and it is positive otherwise

Chunk assigned to each processor's LQ: **756**

Since the two loops are iterated 10 time, we have a total of **7560** chunks statically assigned to each processor's LQ



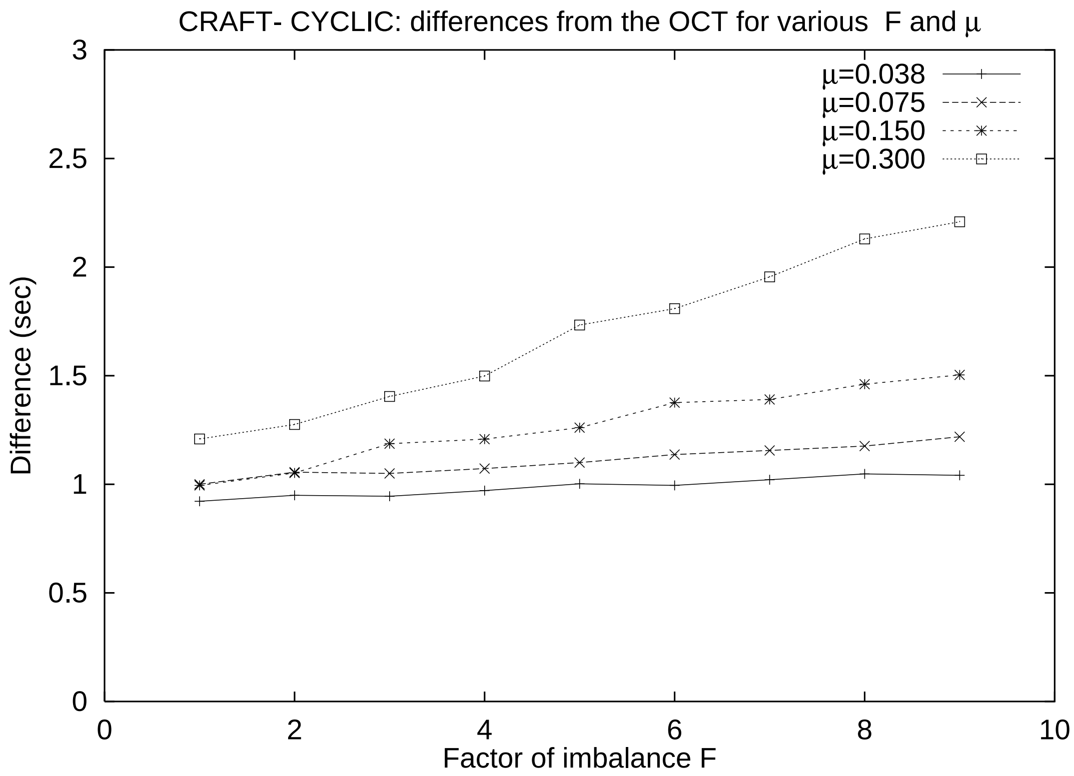
CRAFT Fortran RESULTS: CYCLIC

Differences from OCT for several factors F. Each curve is related to a distinct μ.

The **CYCLIC** distribution is not able to perfectly balance the workload. The imbalance is more considerable for large values of μ

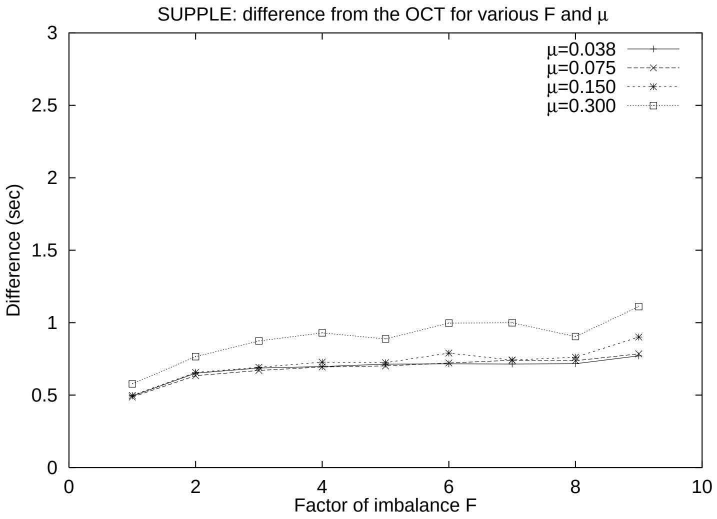
Using a **CYCLIC** distribution, we loose locality in executing the first **Convection( )** loop

We have already a difference of nearly 0.5 sec for $F = 1$ w.r.t. our support. This is, more or less, the cost of using a **CYCLIC** distribution for the balanced case

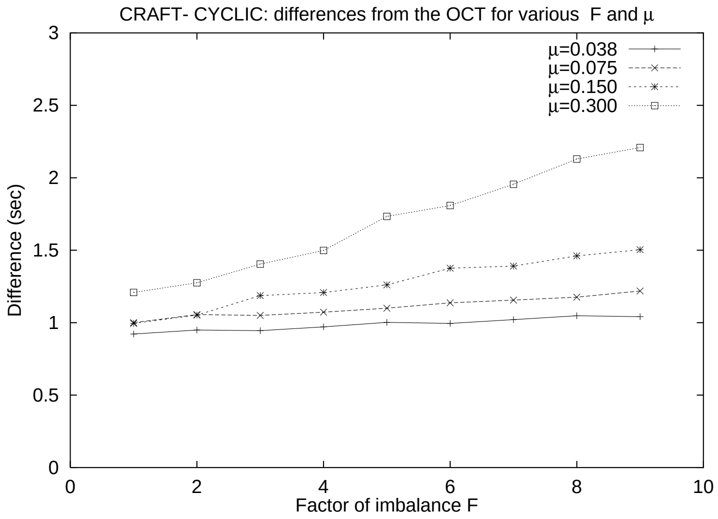


SUPPLE vs. CRAFT: $\mu = 0.038\ msec$

SUPPLE



Fortran CYCLIC

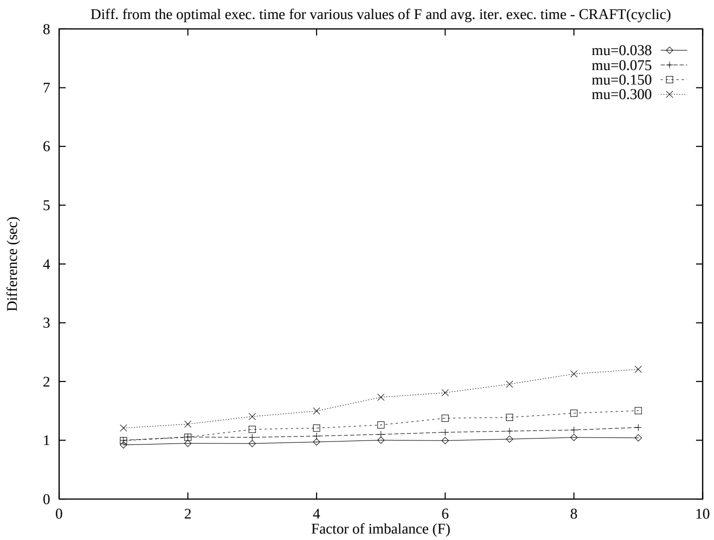


CRAFT Fortran: CYCLIC vs. BLOCK-CYCLIC(4)

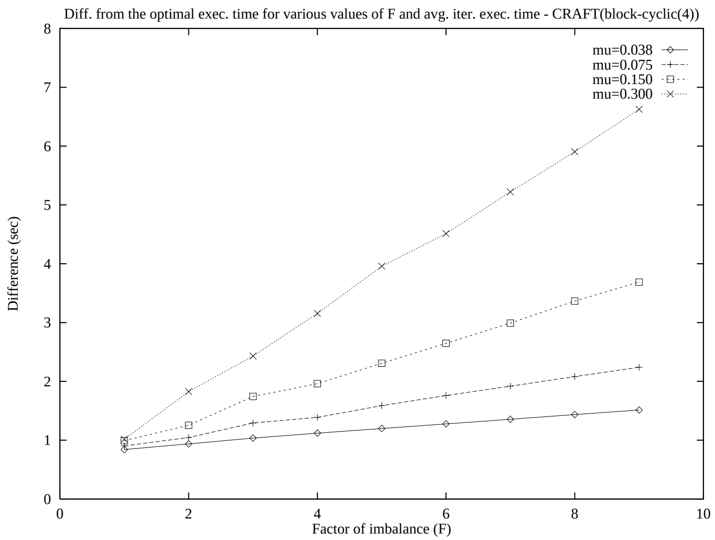
The CRAFT Fortran version **BLOCK-CYCLIC(4)** is worse, except for small values of F.

- In this problem load imbalance is more important than locality: the second loop takes $\frac{3}{4}$ of the total execution time

CYCLIC



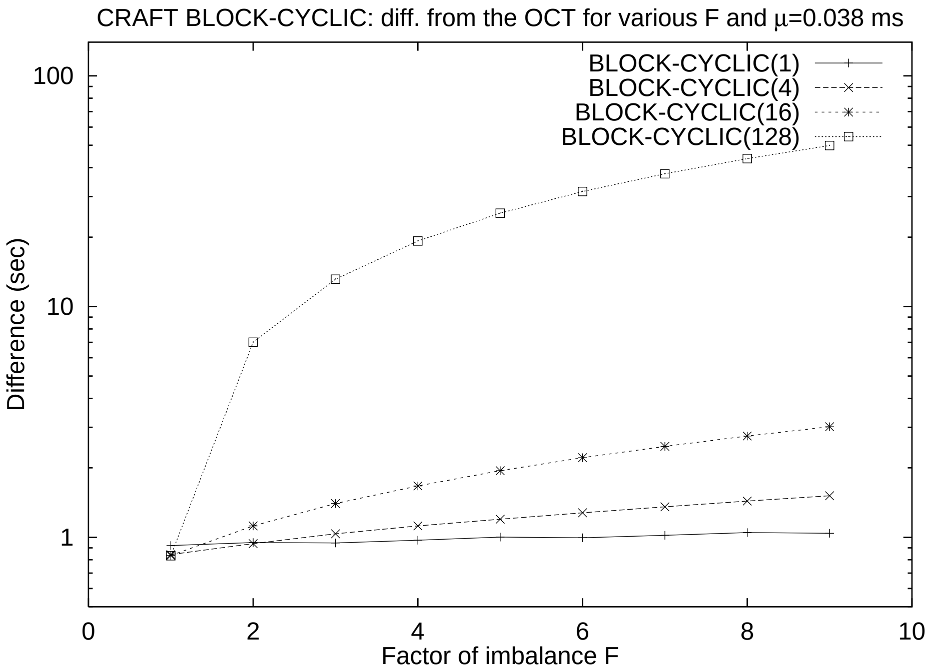
BLOCK-CYCLIC(4)



CRAFT Fortran: from BLOCKING to CYCLIC

Case $\mu = 0.038\ msec$

- Each curve corresponds to a different distribution, and plots the difference from the optimal time (*logarithmic scale*) for various values of F.
- **CYCLIC** is better than **BLOCKING**, except for small values of F.



CONCLUSION AND FUTURE WORK

Our support works well and, due to prefetching and absence of synchronization points, does not introduce too much overhead

Other experiments with other compiler and other machines should be valuable

Open issues:

- Which might be the benefits in using a interrupt-driven messaging layer or active messages ?
- Should our support be used to avoid the adoption of **CYCLIC** distributions?
- Should it be possible to adopt similar dynamic strategies also for run-time supports for other irregular problems?
- Might our support be adopted in a distributed environment (Clusters of WS) where load imbalance may also derive from unpredictable changes in the system load?