Technical Report: TR-95-3, January 18, 1996

On the implementation of the MAP paradigm on a 2D-mesh

Bacci, Bruno Pelagatti, Susanna.

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

Parallelism, Optimal Data Distribution/Collection, P3L This document describes the MAP paradigm of parallelism and the problems related to its efficient implementation on a 2D-mesh. In particular, we first discuss how parallel algorithms exploiting MAP parallelism can be easily expressed by using the P3L Map construct. Then, we discuss an implementation template for a massively parallel architecture with a 2D-mesh topology and Transputer-like processing nodes. The template is asymptotically optimal with respect to the strategies embedded for data distribution, data collection and process allocation.

ACM–CR Subject Classification:

Keyword and phrases: Skeleton Based Parallel Programming, Static Optimisations, Data

UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI INFORMATICA

Technical Report: TR- 3/95

On the implementation of the MAP paradigm on a 2D-mesh

Bruno Bacci and Susanna Pelagatti

Abstract. This document describes the MAP paradigm of parallelism and the problems related to its efficient implementation on a 2D-mesh. In particular, we first discuss how parallel algorithms exploiting MAP parallelism can be easily expressed by using the P3L Map construct. Then, we discuss an implementation template for a massively parallel architecture with a 2D-mesh topology and Transputer-like processing nodes. The template is asymptotically optimal with respect to the strategies embedded for data distribution, data collection and process allocation.

Keywords. Skeleton Based Parallel Programming, Static Optimisations, Data Parallelism, Optimal Data Distribution/Collection, P3L.

February 1995

ADDR:Corso Italia 40,56125 Pisa,Italy. TEL:+39-50-887000 - FAX:+39-50-887226
E_MAIL: susanna@di.unipi.it

1 Introduction

Informally, a parallel algorithm organized according to the MAP paradigm is an algorithm built out of three phases:

- a **distribution** phase, in which the input data X is distributed to a set of *virtual processors*[1];
- a **computation** phase in which all the virtual processors perform a concurrent (and *independent*) computation on the local data,
- a **collection** phase in which all the results produced by the virtual processors are collected and combined in order to build the global result Y of the computation.

In other words, a parallel algorithm exploiting MAP parallelism computes a function f which can be expressed in functional notation as

$$c \circ \alpha g \circ d$$

where

d is the *distribution* (or *scattering*) function, which starting from the input X computes a structure $< x_1, \ldots, x_n >$ in which each x_i includes a subset of the data in X,

α is a second order functional applying a function f to each element of the data structure built by d, that is

$$\alpha g < x_1, \ldots, x_n > = < g(x_1), \ldots, g(x_n) >$$

g is the function computed locally by each virtual processor using the local data x_i, and

c is the *combining* (or *gathering*) function which builds out the real output Y starting from the results $< g(x_1), \ldots, g(x_n) >$.

MAP parallelism is widely used in massively parallel computations achieving high degrees of performance and good efficiency results [5]. Moreover, in most of the cases the functions d, g and c exhibit a regular and simple structure. For instance, the distribution function d can broadcast X to each virtual processor, or can partition X in regular disjoint sets (e.g., element by element if X is a vector, by squared sub-matrices if X is an array and so on).

Our goal is to provide programmers with an efficient and easy-to-use support to many applications exploiting MAP parallelism with a regular structure. This is achieved adopting a template-based organisation of the support [2, 4, 7]. In practice,

[1] Sometimes referred as *processors*, if this does not cause ambiguities.

- the system provides a *skeleton* abstracting the main characteristics of a parallel computation exploiting MAP parallelism, namely a set of virtual processors performing a set of independent computations parametric w.r.t. the three functions d, g and c above. Once programmers have decided to exploit a MAP paradigm of parallelism in their application they have only to instantiate the skeleton with suitable functions.

- the implementation of the MAP skeleton is studied once and for all for the different classes of target architectures by the system designer. For each class, they came up with a set of "efficient" implementation strategies, solving in a standard way problems such as mapping, load balancing, computation granularity, and the like. The strategies devised are then recorded in a parametric network of processes called *implementation template*. Each template is in general parametric both w.r.t. the user defined functions c, d and g and to the amount of resources used (e.g., physical processing nodes). Therefore each template represents a family of process networks implementing the strategies devised. Moreover, we can have different templates for the same class of architectures, because different sets of user defined function could require different implementation choices. Each template is provided with an *analytic performance model* which describes its performance both w.r.t to the characteristics of the user defined code and w.r.t. the characteristics of the architecture

- the support takes care of generating a 'good' instance of one of the templates available once the programmer has instantiated the MAP skeleton.

In this paper, we express the MAP skeleton by using the syntax and semantics of the `p3l` `map` construct [3, 6, 1] and then discuss a new optimised implementation template for architectures with a 2D-mesh interconnection structure using processing nodes with Transputer-like communications.

2 The MAP skeleton in p3l

In `p3l`, the MAP paradigm of parallelism is supplied as a primitive of the language (`map` construct). Programmers need to express only the functions d, c and g according to the `p3l` syntax. As we are interested in MAP computations with a regular structure, there are constraints in the set of d and c functions that can be used. These constraints will be discussed in detail later on.

2.1 `map` definition

The basic structure of a `map` is

```
map pippo in(map-in-list) out(map-out-list)
  body in(body-in-list) out(body-out-list)
end map
```

```
f in(int x,s) out(int y)
  ${
    y = x*s
  }$

map pippo in(int x[10], int s) out(int y[10])
  f in(x[*i],s) out(y[*i])
end map
```

Figure 1: A p3l fragment.

where `pippo` is the name of the `map`, `map-in-list` is the formal parameter list specifying the domain of the function computed by `pippo` and the `map-out-list` is the formal parameter list specifying the codomain. `body` is the name of a p3l module specifying the function g to be applied by each virtual processor, the `body-in-list` is the decomposition function d and the `body-out-list` is the set of virtual processors and the collection function c. The `body` module is written in plain C code. The exact syntax of the p3l `map` construct can be found in [6].

An instance of a p3l `map` is shown in Fig. 1 where `pippo` computes a function f which multiplies all the elements of the input vector `x[10]` by `s` and returns the result in `y[10]`. In this example, 10 virtual processors are defined and the i-th processor computes `y[i]=x[i]`$\times$`s`.

Now, we discuss the meaning of p3l 'free variables' (`*i`,...), and how they can be used to define the virtual processor set, the distribution function d and the combining function c.

2.1.1 Free variables

The variables preceded by a `'*'` in the `body-out-list` and in the `body-in-list` are called *free variables* and are used to express d, c and the set of virtual processors.

A free variable `*i` is a variable which can take all the values of an integer interval $[1, \ldots, n]$. The value of n is equal to the length of the array dimensions in which `*i` appears. In the example, `*i` appears in the first dimension of `x` and `y`, which has 10 elements. In the following, we denote with $m(^*\mathrm{i})$ the upper bound of the range in which `*i` can vary.

2.1.2 Virtual processors

If `*i1,...,*ik` are the free variables appearing in the `body-out-list` and in the `body-in-list` the product

$$m(^*\mathrm{i1}) * \ldots * m(^*\mathrm{ik})$$

gives the number of virtual processors of the map. Each processor in the set can be indexed by a tuple

$$(i_1, \ldots, i_k) \ \text{ where } \ 1 \leq i \leq m(\text{*i1}) \ldots 1 \leq i_k \leq m(\text{*ik})$$

Back to the example of Fig. 1, since the only free variable in the `body-out-list` is `*i` with interval $[1, \ldots, 10]$. We have 10 processors indexed from 1 to 10.

If there were two free variables, say

```
map pippo in(int x[10][5], int s) out(int y[10][5])
  f in(x[*i][*j],s) out(y[*i][*j])
end map
```

we would have 50 virtual processors indexed

$$(i, j) \ \text{ where } \ i = 1, \ldots, 10; j = 1, \ldots, 5;$$

as `*i` ranges in $[1, \ldots, 10]$ and `*j` ranges in $[1, \ldots, 5]$.

The free variables are also used to denote the decomposition and the collection functions (c and d). The c and d's allowed by p3l are a small subset of all the possible decomposition and collection strategies. Nevertheless, they manage to cover most of the regular strategy used in massive parallel computations.

2.1.3 Data decomposition

The *decomposition* of each data structure in input is defined by the position of the free variables within the structure in the `body-in-list`. In practice, if no free variables are specified in any dimensions of a data structure, then the data structure is broadcasted to all the virtual processors. If a free variable, say `*i`, is specified within an array dimension, say `a[][*i][]`, then the element `a[][j][]` will be sent only to those virtual processors for which the index `*i` has value j.

For instance, given the array

```
int a[10][5]
```

and supposing a set of 50 virtual processors indexed

$$(i, j) \text{ where } i = 1, \ldots, 10 \text{ and } j = 1, \ldots, 5$$

we can denote the following data decompositions:

`a[*i][*j]` in which each virtual processor holds a single element of `a`, namely `a[i][j]` (type `int`) is sent to the virtual processor (i, j);

`a[*i][]` in which each virtual processor holds a row of `a`, namely the i-th row of `a` (type `int[10]`) is sent to all the virtual processors $(i, 1) \ldots (i, 5)$;

`a[][*j]` in which each virtual processor holds a column of `a`, namely the j-th column of `a` (type `int[5]`) is sent to all the virtual processors $(1, j) \ldots (10, j)$;

a[][] in which all a (type int[10][5]) is broadcasted to all the virtual processors.

Back to the example in Fig. 1, each virtual processor (i) gets two integers: the scalar s which is broadcasted to all the virtual processors and x[i] as *i has been specified in the first dimension of x in the body-in-list.

If we had the following map instance

```
map pippo2 in(int x[10], int s[5]) out(int y[10][5])
  f in(x[*i],s[*j]) out(y[*i][*j])
end map
```

we would have 50 virtual processors indexed

$$(i, j) \text{ where } i = 1, \ldots, 10 \text{ and } j = 1, \ldots, 5$$

and each virtual processor (i, j) would get x[i] and s[j].

2.1.4 Combining strategies

The combining strategy for each data structure in map-out-list is defined by the position of the free variables within the structure in the body-out-list. The only composition strategy allowed is the one in which each element of the data structure is produced by a distinct virtual processor. In practice, this is equivalent to imposing that all the free variables used in the body-in-list must appear also in any data structure in the body-out-list.

Given this constraints, each element in an output structure corresponding to a given set of values for the free variables is computed by the corresponding virtual processor.

For instance, going back to the Fig. 1 again, y[i] is produced by virtual processor (i), while in

```
map pippo3 in(int x[10], int s[5]) out(int y[5][10],z[10][5])
  f in(x[*i],s[*j]) out(y[*j][*i],z[*i][*j])
end map
```

we have 50 virtual processors (i, j), and each virtual processor (i, j) computes both y[j][i] and z[i][j].

It is also possible to specify a composition strategy in which each virtual processor produces a whole row/column of the output. For instance, the fragment

```
map pippo5 in(int x[10], int s) out(int y[10][5])
  f in(x[*i],s) out(y[*i][])
end map
```

specifies that each virtual processor (i) produces a row of the output structure y.

2.2 How to express some common distribution strategies in p3l

Given the following `map` fragment,

```
map pippo5 in(int s,x[n],z[m],u[n][m],v[n][m]) out(int y[n][m])
  f in(..........) out(y[*i][*j])
end map
```

which defines $n * m$ virtual processors indexed as (i, j), we want to express some common distributions of the input data structures by specifying different expressions in the `body-in-list`.

2.2.1 Broadcast

If we want to broadcast `x` to all the virtual processors we specify

```
  f in(x[],....) out(y[*i][*j])
```

the broadcast of each `x[k]` to all the virtual processors of the row $i = k$ can be stated as

```
  f in(x[*i],....) out(y[*i][*j])
```

if $n = m$ we can also express the broadcast of `x[k]` to all the virtual processors of the column $j = k$ as

```
  f in(x[*j],....) out(y[*i][*j])
```

2.2.2 Cross-product

If we want on each virtual processor a couple `x[i],z[j]` belonging to the cross product among all the elements of `x` and all the elements of `y` we specify

```
  f in(x[*i],z[*j]....) out(y[*i][*j])
```

the same can be done for bigger parts of the input data structures. For instance we can feed each virtual processor (i, j) with the i-th row of `u` and the j-th column of `v` by

```
  f in(u[*i][],v[][*j],....) out(y[*i][*j])
```

2.2.3 Ordered pairing

We can select pair of elements that share the same indices, for instance we can feed virtual processor (i, j) with the element (i, j) of each input data structure

```
  f in(u[*i][*j],v[*i][*j]....) out(y[*i][*j])
```

2.3 Examples of MAP parallelism

In this section, we exemplify the usage of the `map` template formalising a couple of parallel algorithms exploiting MAP parallelism. The complexity and the performance figures achieved by those algorithm will be discussed in detail later on.

2.3.1 Mandelbrot set computation

The Mandelbrot set is the set of points in the complex plane which are quasi-stable when iterated in a function. The most commonly used function is

$$z_{n+1} = z_n^2 + c$$

where the complex values z_0 and c are the points being tested in the complex plane. In practice, for each point of the complex plane c we compute z_n until either the sum of the real and imaginary parts of z_n exceeds four, or the number of iterations n exceeds some fixed maximum *resolution* limit.

The C function used to actually compute the stability of a single point is shown in Fig. 2. This fragment takes a couple of floating point values (`zr`,`zi`) representing a point in the complex plane and an integer value (`resolution`) which states the maximum number of iterations allowed. The two `float` variables `ci` and `cr` hold the initial value of z_0 corresponding to the point in the complex plane being tested. At the i-th iteration the pair (`zr`,`zi`) holds the real and imaginary part of z_i. The function returns a value colour holding either the first value of i for which the magnitude 4.0 has been exceeded or `resolution`-1 if the threshold hasn't been exceeded. The two `float` variables `zrs` and `zis` hold the squares of `zr` and `zi` respectively, to avoid to compute them more than one time.

We can easily exploit MAP parallelism in the computation of the Mandelbrot set on a square of the complex plane by distributing each complex point to a distinct virtual processor, computing the colour of each point in parallel and then collecting all the colours in a two-dimensional array `colour[][]`. This is easily done instantiating the `map` skeleton as in Fig. 3.

The program computes Mandelbrot of a square of `NxN` complex points with vertices $(-1.0, -1.0)$, $(-1.0, 1.0)$, $(1.0, -1.0)$, and $(1.0, 1.0)$. Each point is computed by a distinct virtual processor indexed with `(i,j)` (`i=0...N-1,j=0...N-1`). The sampling within the square is done with distance

```
d=(2.0/(float)(N-1))
```

in order to have the virtual processors on the boarders to compute the function on the edges of the complex square.

`pixel` is a the function g to be computed by each virtual processor. It is basically the same C function shown in Fig. 2 plus an initialisation phase in which the virtual processors `(j,k)` computes the initial values for `zi` and `zr`. The p3l keywords `${` and `}$` encapsulate the part in which plain C code can be used. The definition

```
int pixel (float zr, float zi, int resolution)
{
  float cr = zr;
  float ci = zi;
  float zrs = 0;
  float zis = 0;
  int colour,i;

  for (i=0; i<resolution; i++)
    if (!(zrs+ zis > 4.0))
         {
           zrs =zr *zr;
           zis =zi *zi;
           zi =2.0*zr*zi+ci;
           zr =zrs-zis+cr;
           colour =i;
         }
  return colour;
}
```

Figure 2: A C function used to compute a single pixel

```
pixel in (int resolution, int j, int k) out (int colour)
  ...
end
```

is equivalent to a C functional definition in which the `result` is explicitly stated in the `out` parameter list.

Finally, `mandelbrot` is an instance of the `map` skeleton in which the *distribution* is essentially the cross product of two integer vectors of length `N` initialised with all the possible values of the `i` index of the virtual processors (`index_i`) and of the `j` index of the virtual processors (`index_j`). Namely,

```
index_i = [0,1,2,....,N-1]
index_j = [0,1,2,....,N-1]
```

The cross product generates `NxN` virtual processors indexed with the characteristics discussed above. Each virtual processor `(i,j)` computes `pixel` and outputs the corresponding colour.

```
pixel in (int resolution, int j, int k) out (int colour)
     ${
       float zr,zi,cr,ci,zrs,zis;
       float d=(2.0/(float)(N-1));      /* initialization phase */
       int i;
       zr = -1.0 + (float)(j)*d;
       zi = -1.0 + (float)(k)*d;
       cr = zr;
       ci = zi;
       zrs = 0;
       zis = 0;

       for (i=0; i<resolution; i++)
         if (!(zrs+ zis > 4.0))
               {
                 zrs =zr *zr;
                 zis =zi *zi;
                 zi =2.0*zr*zi+ci;
                 zr =zrs-zis+cr;
                 colour =i;
               };
     }$
end


map mandelbrot in(int index_i[N], int index_j[N], int resolution)
               out(int colour[N][N])
    pixel in(resolution, index_i[*i], index_j[*j]) out(colour[*i][*j])
end map
```

Figure 3: The p3l program for Mandelbrot computation.

```
{
...
  float a[n][n],b[n][n],c[n][n];
  int i,j,k;

  /* reading a,b */
  ...

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

  /* printing the results out */
  ...
}
```

Figure 4: A simple C fragment computing matrix multiplication

2.3.2 Squared Matrix-by-Matrix Multiplication

Suppose to have two $n \times n$ matrices `a` and `b` and to compute the matrix `c` whose elements are given by

$$c(i,j) = \sum_{k=1}^{n} a(i,k) * b(k,j)$$

A straightforward sequential program implementing the previous definition is the one in Fig. 4. Matrix multiply can be computed in parallel exploiting MAP parallelism by defining a set of $n \times n$ virtual processors each computing an element of `c`. More precisely, in the distribution phase each virtual processor (i,j) is provided with the i-th row of `a` and the j-th column of `b` and produces `c[i][j]`. To do this the map skeleton can be instantiated as in Fig. 5. `mul_one_elem` is the function g to be computed by each virtual processor and `mul` is the map instance computing the matrix multiply in parallel according to the algorithm described above.

3 A template for MAP parallelism on a 2D-mesh

An *implementation template* [2, 7] is a parametric network of processes which implements a given skeleton on a particular class of target architectures. The network has a straightforward mapping onto the target architecture and its performance features are modelled

```
mul_one_elem in (int a_row[n], int b_col[n]) out (int c)
     ${
       int k;

       c=0;
       for (k=0; k<n;k++)
         c=c+a_row[k]*b_col[k];

     }$
end


map mul in(int a[n][n], int b[n][n]) out(int c[n][n])
    mul_one_elem in(a[*i][], b[][*j]) out(c[*i][*j])
end map
```

Figure 5: A p3l program for matrix-by-matrix multiply.

by analytic formulae with an acceptable approximation.

We are interested in devising such a template for the `map` skeleton described before onto the class of target architectures having a 2D-mesh interconnection topology and a Transputer-like processing node structure. We assume that:

- before executing the `map` all the input data structures are concentrated on a given node (the *input node*) of the mesh grid and
- after the execution the output data structure is concentrated on a single node (the *output node*).

We want to devise an efficient process network that is able to distribute the input data to the virtual processors (according to d), compute g and collect the results (according to c) in order to achieve the minimum execution time, token into account both the characteristics of the user specified code (g, c and d) and the characteristics of the architecture (e.g., transmission time, computational power of each node etc. . .).

The problems to be solved are basically two

1. deciding how many virtual processors execute on a single physical node. In practice, each physical node will run a *worker* process computing the emulation of the set of virtual processors assigned to that physical node and including some code to cooperate with the other workers for the initial distribution of the data and the final collection.[2] (*deciding the number of workers*);

[2]In the following we will use the term *worker* both to denote the process running on a single physical node and the physical node itself.

2. deciding how to arrange the workers on the mesh topology in order to minimize the time spent for the distribution and the collection of data (*choosing the worker allocation*).

Let us study the allocation problem first, leaving the actual number n of workers used as a parameter (n can vary between 1 and the number of virtual processors needed). This makes sense as n depends on the characteristics of the machine and on the user code in a particular `map` instantiation.

Using a 2D-mesh, it is easy to understand that in order to minimize the distribution/collection time the n workers have to be allocated on a contiguous set of physical nodes. Several different contiguous mappings are possible (e.g. on a line of n processor, on a square of $\sqrt{n}$) and each shape has different characteristics with respect to the cost of distributing and collecting data. However, *it can be proved that the mapping minimising the distribution/collection time is a rectangle with basis and height as close as possible to $\sqrt{n}$, as this shape minimizes the diameter of the area interested by the distribution/collection activity.* This result is discussed in detail in App.A, where we also prove some lower bounds for distribution and collection of data on a mesh network.

In the next Section (3.1) we discuss an implementation template in which the workers are mapped on a square and the distribution/collection algorithms adopted asymptotically reach the lower bounds. Then, in Section 3.2 we present a detailed performance model describing the template. This model will allow us to devise the best worker number n for a given `map` instance and to optimize the resource utilization.

3.1 A squared map template

The implementation template proposed for the `map` skeleton on a 2D-Mesh is built out of the process network sketched in Fig. 6. In the picture, each square corresponds to a distinct process and each line corresponds to a (bidirectional) communication channel. The dotted processes take care of the input and output. The upper left process (with an incoming arrow) is called *emitter* and reads the input data structures from the external environment. The lower right process (with an outgoing arrow) is called *collector* and sends the results off. The rest of processes implement a set of *workers*. Each worker is denoted with W_{ij}, where i and j are respectively the number of row and column where the process is located. Each worker W_{ij} executes a partition of the virtual processes defined within the `map` skeleton. In order to avoid confusion, from now on we will call each virtual process a **task**.

Given an input data X, each worker executes three different phases:

- *distribution*: first, all the workers cooperate to distribute X is among the workers the according to d and to the set of tasks assigned to each worker;
- *computation*: then, each worker autonomously computes its partition of the tasks;
- *collection*: finally, all the workers cooperate to built the result data structure Y according to c.

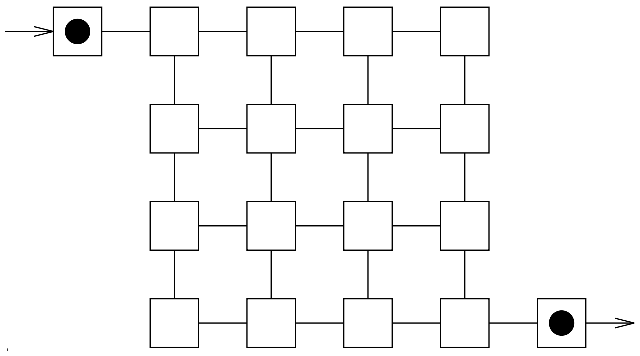

Figure 6: The process structure of the implementation template proposed

The template is parametric with respect to the number of workers n^2 and to the user-defined parts of the `map` skeleton such as the size of data traversing the channels and the computation g to be executed by each virtual processor.

In the following paragraphs, we discuss in detail how the tasks are assigned to the workers (*allocation strategy*) and the algorithms followed in the three phases.

3.1.1 Allocation of the virtual processors (tasks)

The allocation strategy is very simple. Supposing to have k tasks numbered from 1 to k as above and n^2 workers, each computes k/n^2 or k/n^2+1 tasks.

More precisely, suppose that

- the workers are numbered from 1 to n^2 in such a way they the workers of the i-th row range from W_{i+1} to W_{i*n},
- q is the result of the integer division of k by n^2 and,
- r is the *rest* of the integer division of k by n^2.

all the workers compute q tasks, except the last r workers (from W_{n^2-r+1} to W_{n^2}) which computes $q+1$ tasks.

The tasks of a generic `map` instance can be numbered from 1 to k as follows. Suppose to have h free variables `*i1`,...,`*ih` and suppose each variable varying from 1 to $m(*\text{ij})$. The global number of tasks to be allocated is

$$k = m(*\text{i1}) * \ldots * m(*\text{ih})$$

and each one is indexed by an integer tuple

$$(i_1, \ldots, i_h) \;\; \text{where} \;\; 1 \leq i \leq m(*\text{i1}) \ldots 1 \leq i_h \leq m(*\text{ih}) \; .$$

The k virtual processors can be uniquely numbered from 1 to k by assigning to the processors

$$(i_1, \ldots, i_h)$$

the number

$$1 + i_1 * (m(\texttt{*i2}) * \ldots * m(\texttt{*ih})) + i_2 * (m(\texttt{*i3}) * \ldots * m(\texttt{*ih})) + \ldots + i_k$$

or, in a more compact form,

$$1 + \sum_{j=1}^{h} i_j * \prod_{r=j+1}^{h} m(\texttt{*i}_r)$$

As an example, if we have three free variables, say `*i`, `*j` and `*k` with $m(\texttt{i}) = 2$, $m(\texttt{j}) = 3$ and $m(\texttt{i}) = 2$, we have $k = 12$ and

```
*i *j *k
 0  0  0   ->  0*6 + 0*2 + 0 + 1 = 1
 0  0  1   ->  0*6 + 0*2 + 1 + 1 = 2
 0  1  0   ->  0*6 + 1*2 + 0 + 1 = 3
 0  1  1   ->  0*6 + 1*2 + 1 + 1 = 4
 0  2  0   ->  0*6 + 2*2 + 0 + 1 = 5
 0  2  1   ->  0*6 + 2*2 + 1 + 1 = 6
 1  0  0   ->  1*6 + 0*2 + 0 + 1 = 7
 1  0  1   ->  1*6 + 0*2 + 1 + 1 = 8
 1  1  0   ->  1*6 + 1*2 + 0 + 1 = 9
 1  1  1   ->  1*6 + 1*2 + 1 + 1 = 10
 1  2  0   ->  1*6 + 2*2 + 0 + 1 = 11
 1  2  1   ->  1*6 + 2*2 + 1 + 1 = 12
```

Supposing the tasks are to be scheduled on 9 workers we will have the following scheduling

```
workers  w1   w2   w3   w4   w5   w6   w7    w8     w9

task(s)  1    2    3    4    5    6    7,8  9,10   11,12
```

Using the numbering described, the compiler can easily compute the tasks to be computed by each worker. In particular, it can deduce the number of the first task to be computed (`first_task[i]`) and the number of tasks to be computed (`num_task[i]`). These values are used within the worker code to control the computation phase.

3.1.2 Data distribution strategy

The data distribution is accomplished by means of the **wormhole broadcast** algorithm which broadcasts the *whole* input structure X to all the workers *without conflicts*. Suppose the length in bytes of the input data structure X is m, the algorithm works as follows:

1. X is split in m/d blocks of length d by the emitter process which send them one after the other to W_{11};

2. as soon as W_{11} gets a new block from the emitter it sends the block both along the first row (to W_{12}) and along the first column (to W_{21});

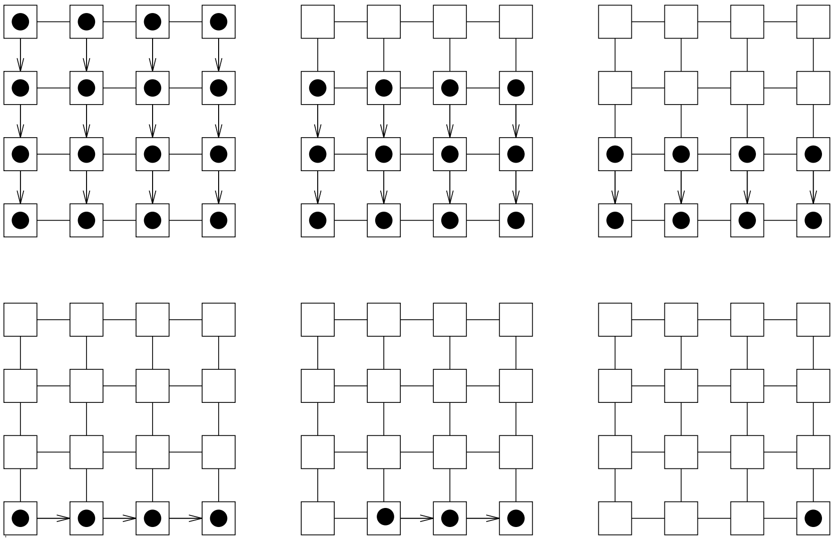

Figure 7: A simple collection strategy

3. each worker W_{1j} (on the first row) gets a sequence of m/d blocks from $\mathsf{W}_{1(j-1)}$ and it sends each block both along the first row (to $\mathsf{W}_{1(j+1)}$) and along the j-th column (to W_{2j});
4. each worker W_{ij} (on row $i = 2, \ldots, n-1$) gets a new block from $\mathsf{W}_{(i-1)j}$ and sends it off along the j-th column (to $\mathsf{W}_{(i+1)j}$);
5. each worker W_{in} on the last row simply gets blocks from $\mathsf{W}_{i(n-1)}$.

After receiving m/d blocks each worker has finished the distribution phases and can start computing. The number of communication required is $o(n + \frac{m}{d})$ as it takes $o(n)$ to wait until the first block reaches the farthest worker W_{nn} and $o(m/d)$ to wait for the rest of the sequence.

3.1.3 Computation strategy

Once the data have been distributed, each worker executes the subset of virtual processors allocated to it. This is accomplished by means of a loop that starts from the first virtual processor to be computed by the worker according to the allocation rule (`first_task` computed by the compiler and performs `num_task` iterations). At each iteration the worker basically

1. prepares the slice of data needed by the task in execution in a suitable format and
2. calls the user defined code to perform the function g on the slice.

3.1.4 Collection strategy

The template first collects all the results along the columns of the worker square and then we collect along the last row and in the mean time we output the results obtained towards the collector process.

The collection strategy is depicted in Fig. 7. Where we have 16 workers and the results are collected in the worker at the lower right corner. In the picture, the black dots signal when a node has got results to be collected and the arrows show the communications to take place at the next step of the collection algorithm. At the beginning all the workers have some results to deliver to the collecting node. Then all the results are collected along the columns by using only communications between neighbours in the same column. After all the results of column j have been collected in the j-th worker of the last row we start gathering results along the last row with the same strategy. The number of communications needed by this simple algorithm is $o(m)$.

3.1.5 Template instantiation

Summarizing, the map implementation template depends on the following parameters

- data types in the `in` and `out` lists of the `map` instance, which determines the amount of data to be sent on the template communication channels and the length of the distribution/collection phases;
- the user code defining the function g to be computed by each worker;
- the number of workers to be included;
- the granularity of the blocks sent during the distribution phase.

When the template is used to build an implementation of *a specific instance* of the `map` skeleton, all those parameters need to be instantiated. The first two are fixed by the instance of the map skeleton itself as they can be directly derived from the `in` and `out` lists and from the `body` sequential part.

On the other hand, fixing n and d is more difficult as they strongly impact on the performance and on the efficiency achieved by the template itself. However, it is possible to tune n and d at compile time by means of analytic performance models describing the behaviour of the completion time of the template.

We will discuss this point in a great detail in the following section.

3.2 Analytic performance formulae

Given an instance of the `map` skeleton, it is possible to choose the number of workers n^2 included in the template and the broadcast 'granularity' d which allow the minimum completion time to be achieved.

This is done by deriving an analytic formulation for the completion time as a function of both n and d and approximating the minimum point. Moreover, as beyond a certain number of workers the completion time slightly decreases while efficiency drops down, it is possible to implement a simple heuristics to reduce the number of workers at a price of a modest slowdown. Details on this are given in App. B.

3.2.1 Parameters

m length (in bytes) of the data structure to be computed;

d broadcast granularity (length in bytes of each packet during the 'wormhole' broadcast);

n side of the square of workers (n^2 workers);

k total number of tasks to be computed;

s length in bytes of the data structure produced by each task;

t_w execution time of a task (sequential);

t_0 startup time of a communication;

t_1 transmission time (per byte);

3.2.2 Formulae

The **distribution time** spent for the initial broadcast T_d is given by the 'latency of the worm'—that is the time spent by the first packet to reach worker W_{nn}— and by the steady state time —in which the rest of the packets arrives. The latency is the sum of the following quantities:

- $t_0 + m * t_1$, which is the time spent by the emitter E to read the data structure from the previous stage;
- $t_0 + d * t_1$, to send the first packet to the worker W_{11};
- $(3 * t_0 + d * t_1) * (n - 1)$ to pass the first row of workers;
- $(2 * t_0 + d * t_1) * (n - 1)$ to reach the workers of the last row (in parallel).

After the initial latency has passed by, the remaining $(\frac{m}{d} - 1)$ packets reach W_{nn} every $(3 * t_0 + d * t_1)$. Thus the total distribution time is given by:

$$T_d(n, d) = 2 * t_0 + (m + d) * t_1 + (5 * t_0 + 2 * d * t_1) * (n - 1) + (\frac{m}{d} - 1) * (3 * t_0 + d * t_1)$$

The **collection time**, spent to collect all the `out` data structure is given by:

$$T_r(n) = (n - 1) * t_s + (\frac{k(n - 1)}{n^2}) * t_s + k * t_s$$

where $t_s = t_0 + s * t_1$.

The **computation time** is

$$T_c(n) = \frac{k * t_w}{n^2}$$

Thus, the **completion time** of the squared map template is given by:

$$T(n,d) = T_d(n,d) + T_r(n) + T_c(n) \tag{1}$$

3.2.3 Optimizing the completion time

Optimizing the completion time (1) means finding the values of d and n for which $T(n,d)$ is minimal.

If n is fixed, we can find the optimal granularity by deriving $T(n,d)$ w.r.t. d and equating the derivative to zero. Namely,

$$\begin{aligned}\frac{\partial T(n,d)}{\partial d} &= t_1 + 2*(n-1)*d*t_1 - \frac{3*t_0*m}{d^2} - t_1 \\ &= 2*(n-1)*d*t_1 - \frac{3*t_0}{d^2}\end{aligned}$$

Thus $\frac{\partial T(n,d)}{\partial d}$ is equal to 0 only if

$$d = \sqrt{\frac{3*t_0*m}{2*(n-1)*t_1}} \tag{2}$$

If we want to optimize $T(n,d)$ both respect to n and respect to d we derive both w.r.t. to n and d and solve the corresponding system of equations. Namely,

$$\frac{\partial T(n,d)}{\partial n} = 5*t_0 + 2*d*t_1 - \frac{2*k}{n^3}(t_w - t_s) - \frac{k*t_s}{n^2} + t_s \tag{3}$$

By substituting 2 in 3 we obtain a non-linear equation in n which can be approximated with the usual methods.

3.2.4 Experimental results

In order to validate the model described above, we show some experimental figures for the Mandelbrot set computation and the Matrix Multiplication.

Mandelbrot The results obtained with the Mandelbrot set program with a resolution of 300 and a matrix 300×300 are shown in Fig. 8, where the x-axis correspond to the number of workers used within the map template and the y-axis correspond to the completion time of the program expressed (in μs). The line marked with diamonds correspond to the analytic model of the completion time T_c. The black circles correspond to the completion

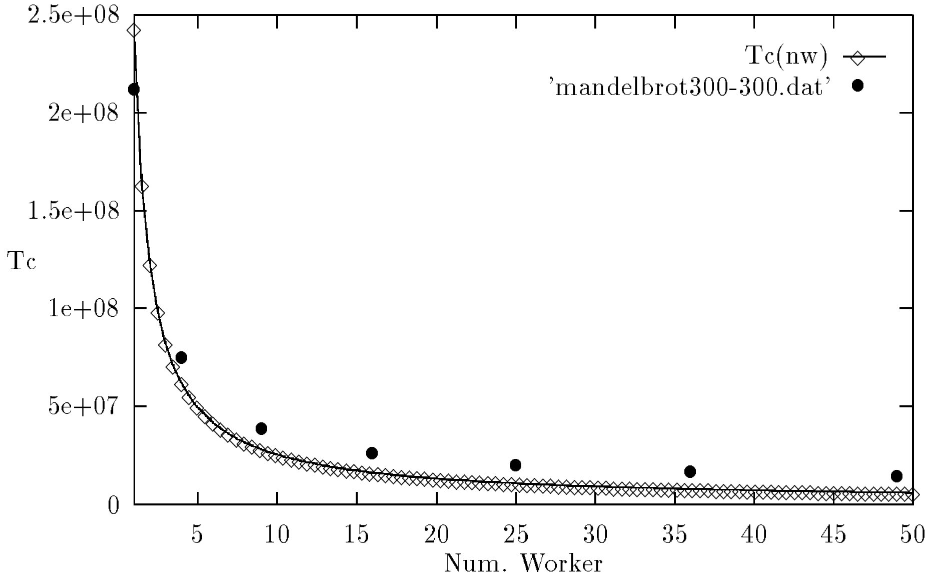

Figure 8: Mandelbrot set experiment

time experimentally measured for the real template running onto a Meiko CS1 with 52 Transputer nodes.

The cost to execute a worker process is $t_w = 2191\mu s + 500\mu s$ where $2191\mu s$ is the time spent in the execution of the user specified code and $500\mu s$ is the housekeeping (preparation of the parameters, call of the user defined code and so on). Moreover, $m = 2 * 300 * 4$ (2 vectors of 300 `float` each), $s = 4$ (two float), $d = 800$, $k = 300 * 300$, $ts = t_0 + s * t_1$ (where $t_0 = 6.9$ and $t_1 = 1.1$).

The results obtained with a matrix 400×400 and a resolution of 500 are shown in Fig. 9. In this case, $m = 2 * 400 * 4$, $t = 3623\mu s + 500\mu s$, $s = 4$, $d = 800$ $k = 400 * 400$ and t_s, t_1 and t_0 are the same as above.

The exact values of the completion time measured in our experiment are shown in C.1.

Matrix multiplication The results obtained running the matrix multiply program with two 200×200 matrices are shown in Fig. 10. The x-axis correspond to the number of workers in the map template, the y-axis correspond to the completion time T_c as usual.

The plotting of the analytic formula for T_c (diamond plot) is obtained with the following parameter values: $m = 2 * 200 * 200 * 4$, $t = 224000\mu s$, $s = 4$, $d = 800$ $k = 200$ and t_s, t_1 and t_0 the same as above.

References

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}l: A struc-

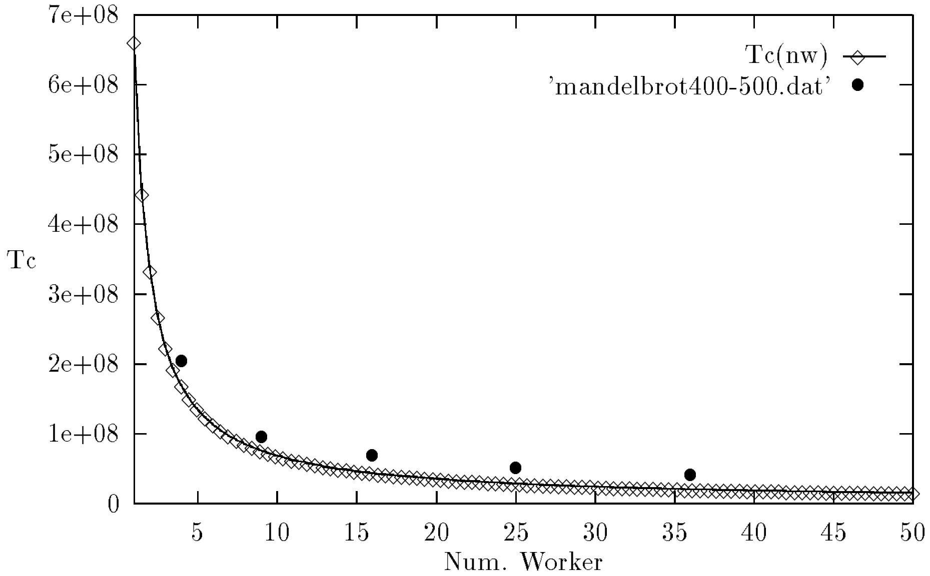

Figure 9: Mandelbrot set experiment

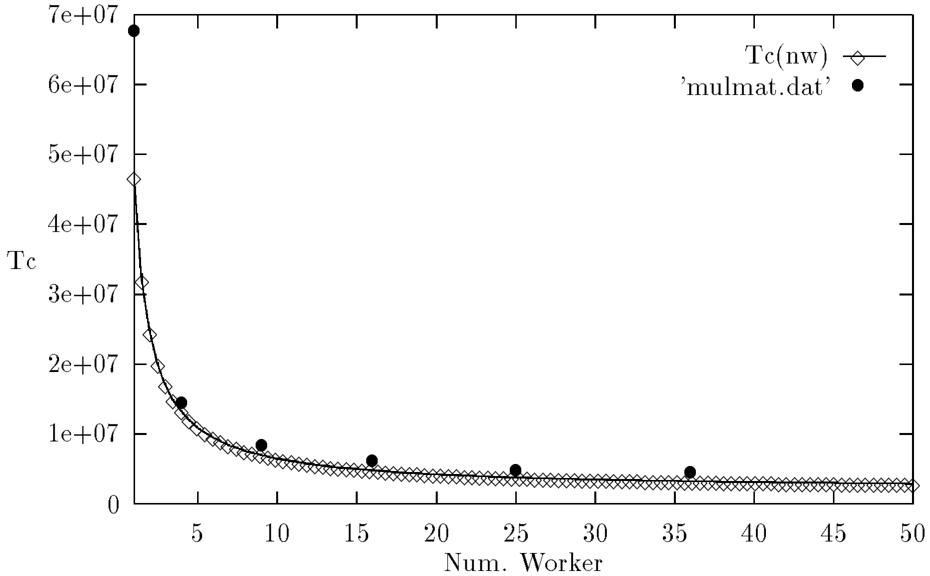

Figure 10: Mulmat experiment

tured high level programming language and its structured support. Technical Report TR 36/93, Department of Computer Science, University of Pisa (Italy), 1993. To be published on *Concurrence: Practice and Experience*.

[2] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, Massachusetts, 1989.

[3] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, August 1992.

[4] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, R.L. While, and Q. Wu. Parallel Programming using Skeleton Functions. Unpublished note, Dept. of Computing, Imperial College, London., 1992.

[5] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processor*. Prentice-Hall, 1988.

[6] S. Orlando. The P^3L language manual (current implementation). Technical Report HPL-PSC-92-52, Hewlett Packard Laboratories, Pisa Science Center (Italy), November 1992.

[7] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. PhD thesis, TD-11/93, Department of Computer Science, University of Pisa (Italy), 1993.

A Optimality on the squared worker mapping and lower bounds of distribution and collection

Due to the restrictions imposed by the p3l `map` on the distribution function d, all the data structures in input but arrays are sent in broadcast to all the logical workers. On the other hand, for an array input structure `a[n][p][q]` we can have the following three cases

1. it can be sent in broadcast to all the workers (this happens if `a` is specified as

   ```
   a[][][]
   ```

   in the `body-in-list`);

2. it can be partitioned and each partition sent to a different logical worker (this happens if `a` is specified by using all the free variables, that is we have only two free variables `*i`, `*j` and `a` is specified as

   ```
   a[*i][*j][]
   ```

   in the `body-inlist`);

3. it can be partitioned and each partition sent to a subset of workers (this happens if only a subset of all the free variables is specified within the dimensions of `a`, for instance if there are two free variables `*i` and `*j` and `a` is specified as

   `a[*i][][]` or `a[][][*j]`

   in the `body-inlist`).

Let us suppose that our physical workers are mapped on a rectangle of processing nodes of base b and height h and discuss the cost of distributing `a` in the three cases. We suppose `a` has length m.

In case (1), the broadcast of `a` takes at least $o(m)$ communications to get out from the node holding the input structures (we can use at most 4 links in parallel, so we need at least $m/4$ communications). Moreover, we have to reach all the nodes in the rectangle, which will take in general a number of hops of the same order of the diameter of the rectangle (which is $h + b$).

Thus a lower bound for the broadcast of `a` is $o(m+(b+h))$. It is worth noticing that for a given number of workers $n = k^2$ the minimum of $b + h$ is given by $h = b = k$. In general, if n is not a perfect square, $h + b$ is minimal for values of b and h as close as possible to $\sqrt{n}$.

In case (2), scattering all the partitions of `a` takes at least $o(m)$ to get out from the input node. This bound can be reached by simply outputting first the partitions of `a` with the farthest destination. When we output the last partition we will have reached all the workers as m is in general $\leq n$.

For instance, let us suppose $m = n = k^2$, the workers arranged on a square of side k and the input node `I` be a vertex of the square. The situation is illustrated in the upper left square of 16 nodes shown in Fig. 11. A simple scattering strategy is the following: first `I` fills each worker of the first row with the partitions corresponding to row k. When each worker $W(1, i)$ belonging to the first row has got the partition for worker $W(k, i)$, all the partitions are sent down the columns and the workers of the first row are filled with the partitions corresponding to row $k - 1$. After $k^2 = n$ steps all the workers have received the corresponding partition. Fig. 11 shows some snapshots of the scattering on a 16 nodes square. The black dots stand for partitions and the arrows show the communications that will be sent in the following step.

In case (3), suppose `a[p][q]` and we have 2 free variables, say `*i`, `*j`. If only one free variable is specified within `a`, say `a[*i][]`, then all the workers corresponding to the same value of `*i` will get the same row of `a`. More precisely, we can divide all the workers in p subsets

$$\mathsf{W}_i = \{(i, 1), \ldots, (i, q)\}$$

and each set W_i gets exactly q elements of `a` corresponding to `a[i][]`.

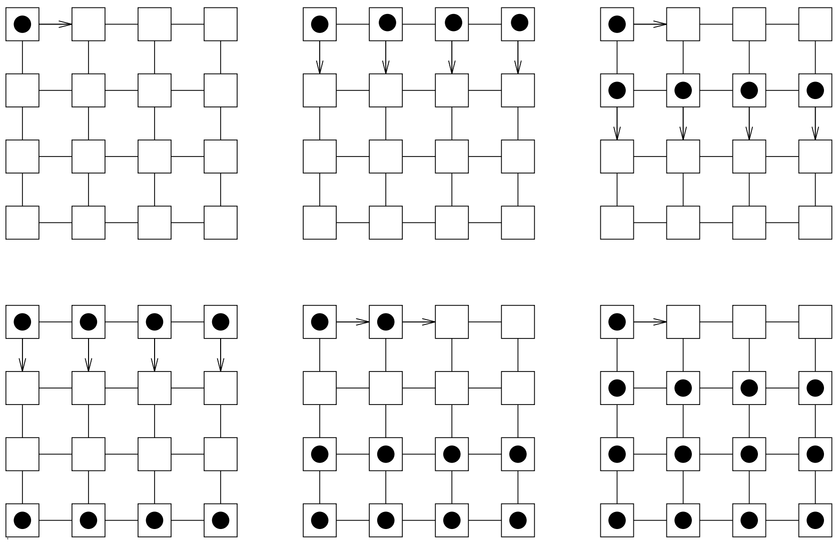

Figure 11: A simple scattering strategy

This kind of distribution takes at least $o(m)$ communications to get `a` out from the input node. Moreover, as the farthest processing node is at a distance $b + h \geq 2 * \sqrt{n}$ another lower-bound to the communication during the distribution phase is given by $o(\sqrt{n} + p)$. Combining the two results, we can state that

- a lower bound for any distribution algorithm is given by $o(\max\{m, b + h + p\})$ and
- the worker mapping that allows the minimum lower-bound is again the square (for which $h + b = 2 * \sqrt{n}$).

Summarizing the three cases, the minimal bounds on the communication need in the distribution phase are achieved for a squared mapping of the physical workers. Moreover, the bounds found for case (1) and (3) can both be reached by using the *wormhole broadcast algorithm* discussed in Section 3.1.

Finally, we can obtain similar results also for the collection phase. According to the restrictions imposed in the `body-out-lists` of a `p3l map` construct each logical worker produces one and only one element of a given output data structure. Thus, if the logical workers are m, we need at least $o(m)$ communications to collect all their results on a single node. If all the workers are arranged on a square and we suppose to collect all the results in a single worker on the square side, this lower bound can be easily reached with the collection algorithm shown in Fig. 7.

B Details on the squared template implementation

B.1 The approximation algorithm for n and d

The algorithm used to find out the optimal values for n and d within the p3l compiler basically approximate the zeros of the derivate of the completion time $T_c(n)$. The derivative is crescent and tends to $-\infty$ when $n = 1$. Thus, the compiler computes the value of the derivative for $n = 1, 2, 3 \ldots$ until it finds a positive value n^*. The optimal value for the square side is $n^* - 1$.

B.2 Finding a good trade-off between speed-up and efficiency

The completion time T_c is such that its value decreases quickly when the number of workers n stays below an 'efficiency saturation point' $\overline{n}$. Beyond $\overline{n}$ the completion time continue to decrease but with a very slow ratio while the efficiency drops down. Within the p3l compiler we look for the point $\overline{n}$ in order to pick up the square side which gives the best trade-off between speed-up and efficiency.

$\overline{n}$ is given by the value of n for which the second derivative of the completion time is equal to zero (i.e., the point in which the concavity of the curve changes) and it is approximated iteratively starting from the optimal value n^* and decreasing n until the value of T_c'' changes sign.

C Experimental results – Tables

C.1 Mandelbrot set

C.1.1 100X100 matrices – Resolution 100

Worker number	Completion time	Speedup
1		–
4	3.01s	
9	2.06s	
16	1.58s	
25	1.36s	
36	1.23s	

C.1.2 300X300 matrices – Resolution 300

Worker number	Completion time	Speedup
1	212.13s	–
4	74.89s	
9	38.80s	
16	26.05s	
25	20.16s	
36	16.59s	
49	14.54s	

Cost of the sequential worker computation $t = 2191\mu s$

C.1.3 300X300 matrices – Resolution 400

Worker number	Completion time	Speedup
1	245s	–
4	97.18s	
9	48.84s	
16	32.61s	
25	24.04s	
36	19.37s	

Cost of the sequential worker computation $t = 2896\mu s$

C.1.4 400X400 matrices – Resolution 500

Worker number	Completion time	Speedup
1		–
4	203.05s	
9	94.82s	
16	67.97s	
25	50.54s	
36	40.42s	

Cost of the sequential worker computation $t = 3623\mu s$

C.2 Matrix multiply

C.2.1 50X50 matrices

Worker number	Completion time	Speedup
1	1.2s	–
4	0.34s	3.52
9	0.277s	4.4
16	0.25s	4.8

The compiler fixes a broadcast grain $d = 436$ bytes and 4 workers. The sequential time of a worker is $t = 11895\mu s$. By varying the broadcast granularity d we observed the following effects on the completion time and on the speedup (with 4 workers).

Grain size	Completion time	Speedup
1	0.942s	1.27
50	0.39s	3.07
200	0.359s	3.33
600	0.350s	3.4
1000	0.352s	3.4
2000	0.358s	3.4
2500	0.36s	3.4

From which we can see that the effects of broadcast granularity are bigger for very small or very big grain values, whereas for a large ranges in between the template behaviour is largely independent of it. Workstation 0.03sec (compl time).

C.2.2 100X100 matrices

Worker number	Completion time	Speedup
1	9.65s	–
4	1.85s	5
9	1.283s	7.5
16	1.06s	9.1
25	0.93s	10.37
36	0.91s	10.6

The compiler fixes a broadcast grain $d = 616$ bytes (154 integers) and 9 workers. The sequential time of a worker is $t = 44265\mu s$. By varying the broadcast granularity d (with 9 workers).

Grain size	Completion time	Speedup
4	3.66s	2.67
100	1.37s	7
1000	1.37s	7
5000	1.32s	7.3
10000	1.36s	7
40000(all)	1.62s	5.9

From which we can see that the effects of broadcast granularity are bigger for very small or very big grain values, whereas for a large ranges in between the template behaviour is largely independent of it. Workstation(HP730) 1.3sec (compl time).

C.2.3 150X150 matrices

Worker number	Completion time	Speedup
1	32.9s	–
4	5.45s	6
9	3.37s	9.76
16	2.68s	12.2
25	2.28s	14.42
36	2.18s	15

The compiler fixes a broadcast grain $d = 752$ bytes (188 integers 120 separate sends) and 16 workers. The sequential time of a worker is $t = 100353\mu s$. By varying the broadcast granularity d (with 16 workers).

Grain size	Completion time	Speedup
4	8.0s	4.1
100	2.9s	11.34
22500	2.91s	11.34
90000(all)	3.61s	9.11

From which we can see that the effects of broadcast granularity are bigger for very small or very big grain values, whereas for a large ranges in between the template behaviour is largely independent of it. Workstation(HP730) 4.8sec (compl time).

C.2.4 200X200 matrices

Worker number	Completion time	Speedup
1	77.8s	–
4	12.26s	
9	7.51s	
16	5.72s	
25	4.8s	16.2
36	4.5s	

The compiler fixes a broadcast grain $d = 740$ bytes and 25 workers. The sequential time of a worker is $t = 178059\mu s$. By varying the broadcast granularity d (with 16 workers).

Grain size	Completion time	Speedup
4	14.4s	
100	5.3s	
40000	5.38s	
160000(all)	6.8s	12.9

From which we can see that the effects of broadcast granularity are bigger for very small or very big grain values, whereas for a large ranges in between the template behaviour is largely independent of it. Workstation(HP730) 10.6sec (compl time).