Exploiting Vector Extensions to Accelerate Time Series Analysis

Ricardo Quislant Ivan Fernandez Eduardo Serralvo
Eladio Gutierrez Oscar Plata

Department of Computer Architecture
University of Malaga (Spain)
quislant@uma.es

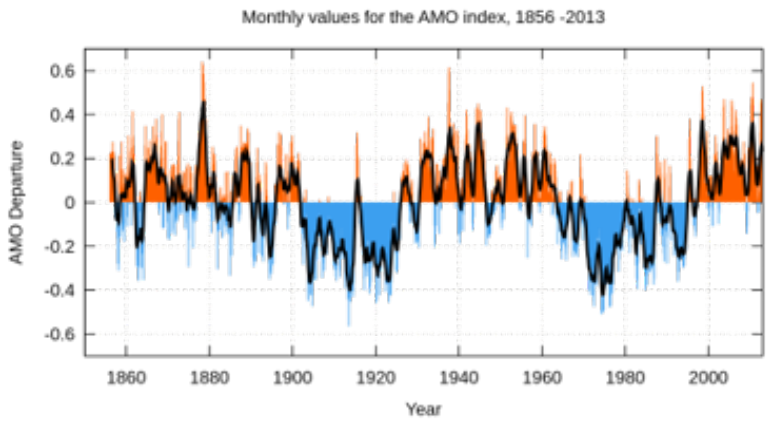
XXXIII Jornadas de Paralelismo

Contenido

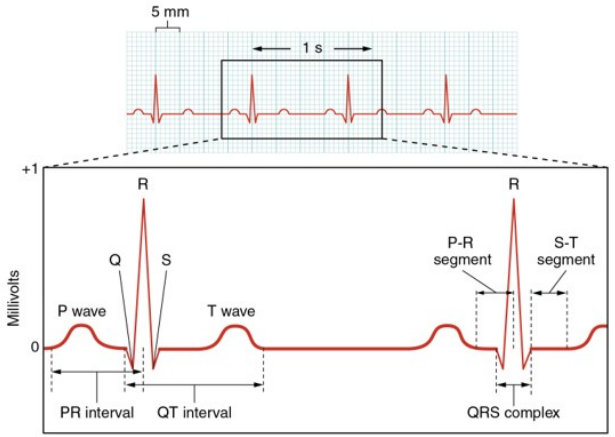
Contenido

Análisis de Series Temporales

Gran interés en multitud de campos

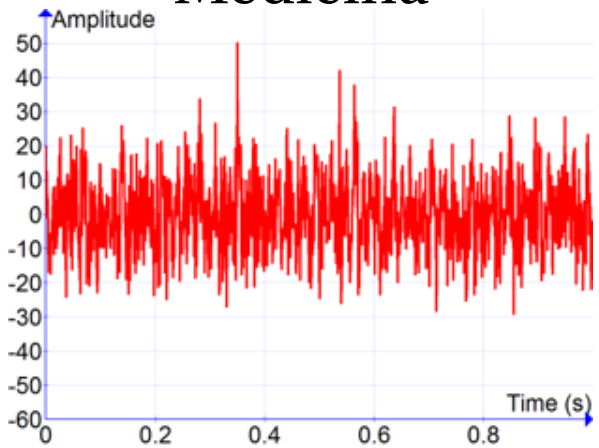


Cambio climático



Medicina

Economía

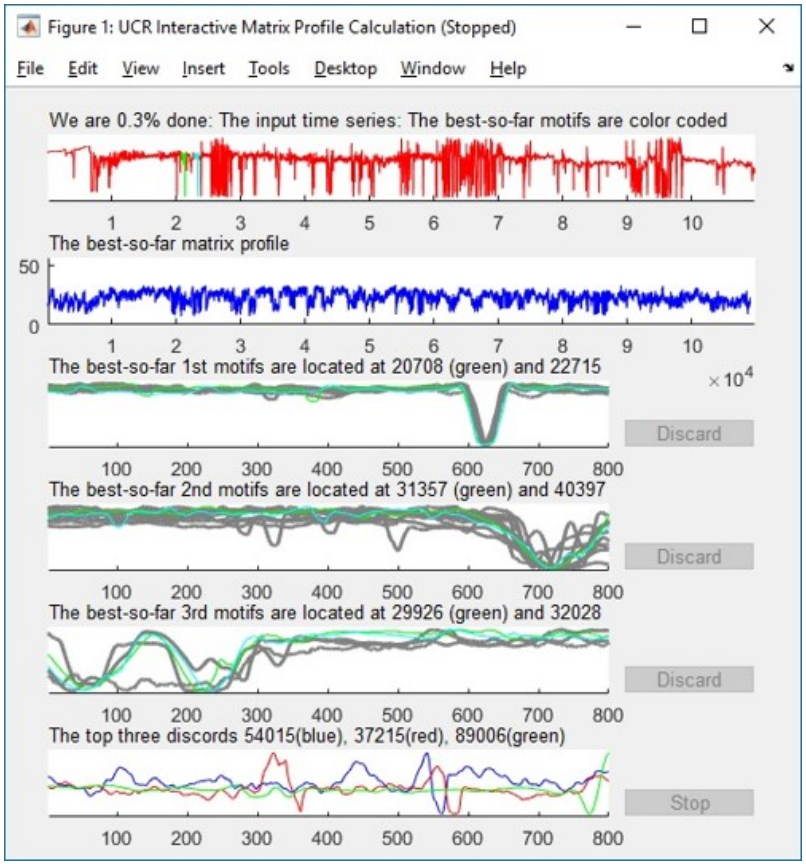


Procesado de señal

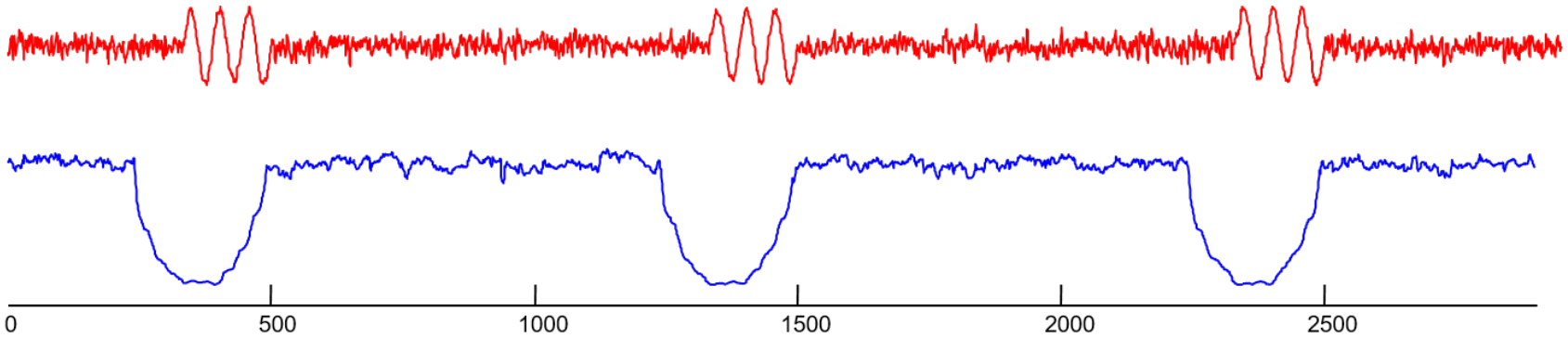
Matrix Profile

https://www.cs.ucr.edu/~eamonn/MatrixProfile.html

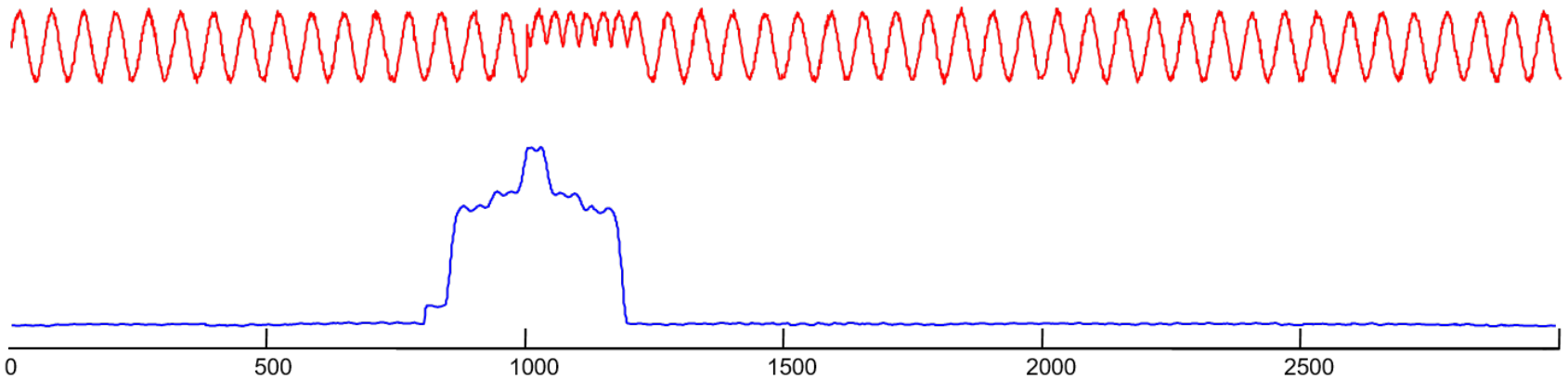
- Herramienta *open source* para descubrir **motifs** (similitudes) y **discords** (anomalías) en series temporales
- Implementado en varios lenguajes: C++, Python, CUDA, R, MATLAB
- No necesita umbral de similitud



Motifs y Discords



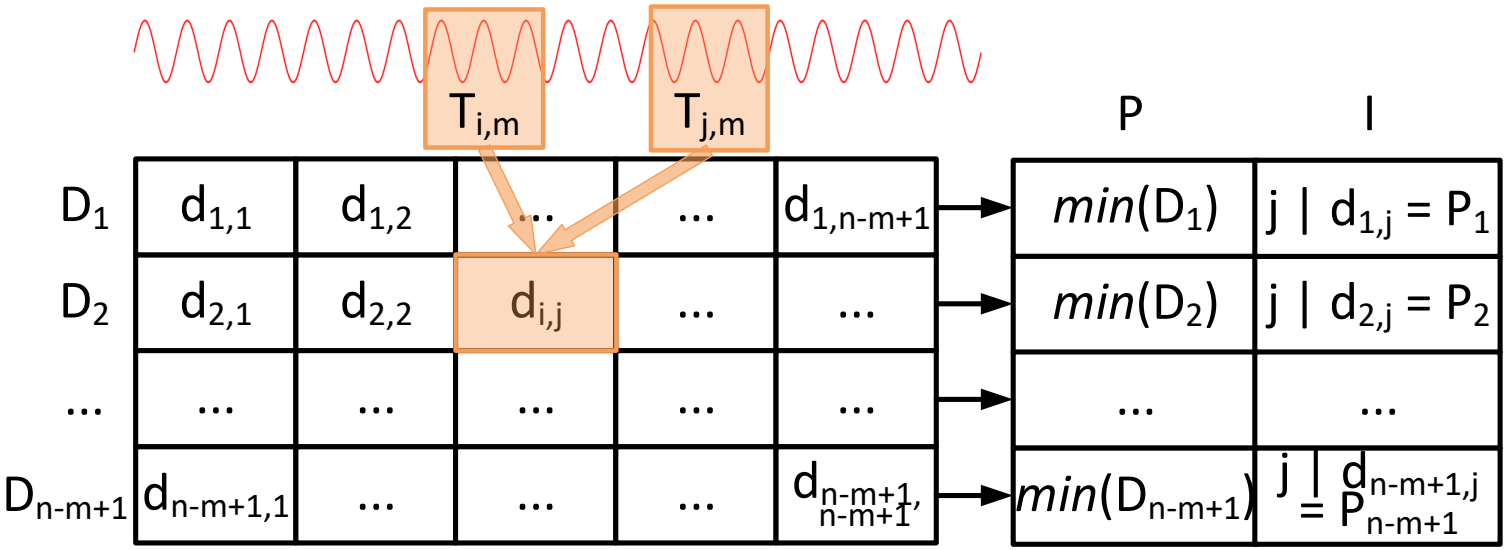
Ejemplo sintético de **similitud** (motif)



Ejemplo sintético de **anomalía** (discord)

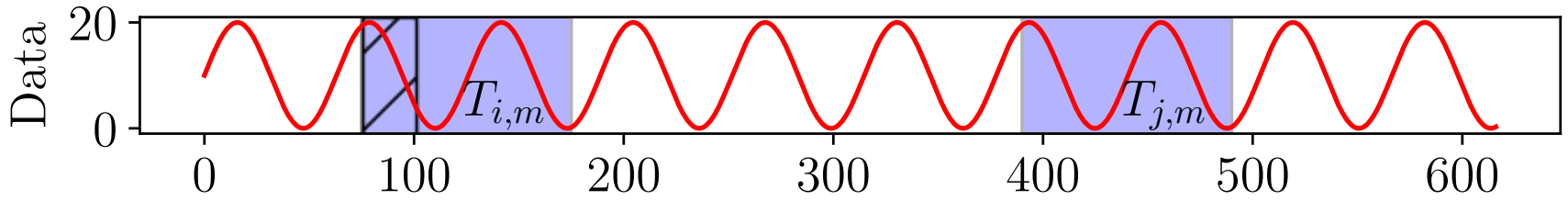
Cálculo del Profile

El profile P y su vector de índices I se calcula al vuelo usando la matriz:



- Los algoritmos de Matrix Profile **no alamacenan la matriz** en memoria
- Las celdas de la matriz se calculan por **diagonales** para reutilizar el cómputo del producto escalar entre subsecuencias
- Implementaciones más importantes: **SCRIMP** y **SCAMP**

SCRIMP



SCRIMP comprueba la similiridad de **cada subsecuencia de una serie temporal con todas las demás**. Para ello, se calcula la distancia (Euclídea) usando la siguiente ecuación:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{Q_{i,j} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right)}$$

donde $Q_{i,j}$ es el producto escalar de las subsecuencias de longitud m, y μ_i, μ_i, σ_i, σ_j son sus respectivas medias y desviaciones típicas

SCAMP

SCAMP sigue un esquema de computación similar a SCRIMP, pero **reemplaza el producto escalar por una suma de productos centrados en la media** para computar el coeficiente de correlación de Pearson:

$$df_i = \frac{T_{i+m-1} - T_{i-1}}{2}$$

$$dg_i = T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}$$

Este coeficiente se puede calcular con menos operaciones y tiene menos error de redondeo de coma flotante que SCRIMP, pudiendo obtener de él la distancia euclídea en O(1):

$$D_{i,j} = \sqrt{2m(1 - P_{i,j})}$$

Motivación

- Los algoritmos para computar el Matrix Profile son **embarrassingly parallel** y pueden beneficiarse de la vectorización

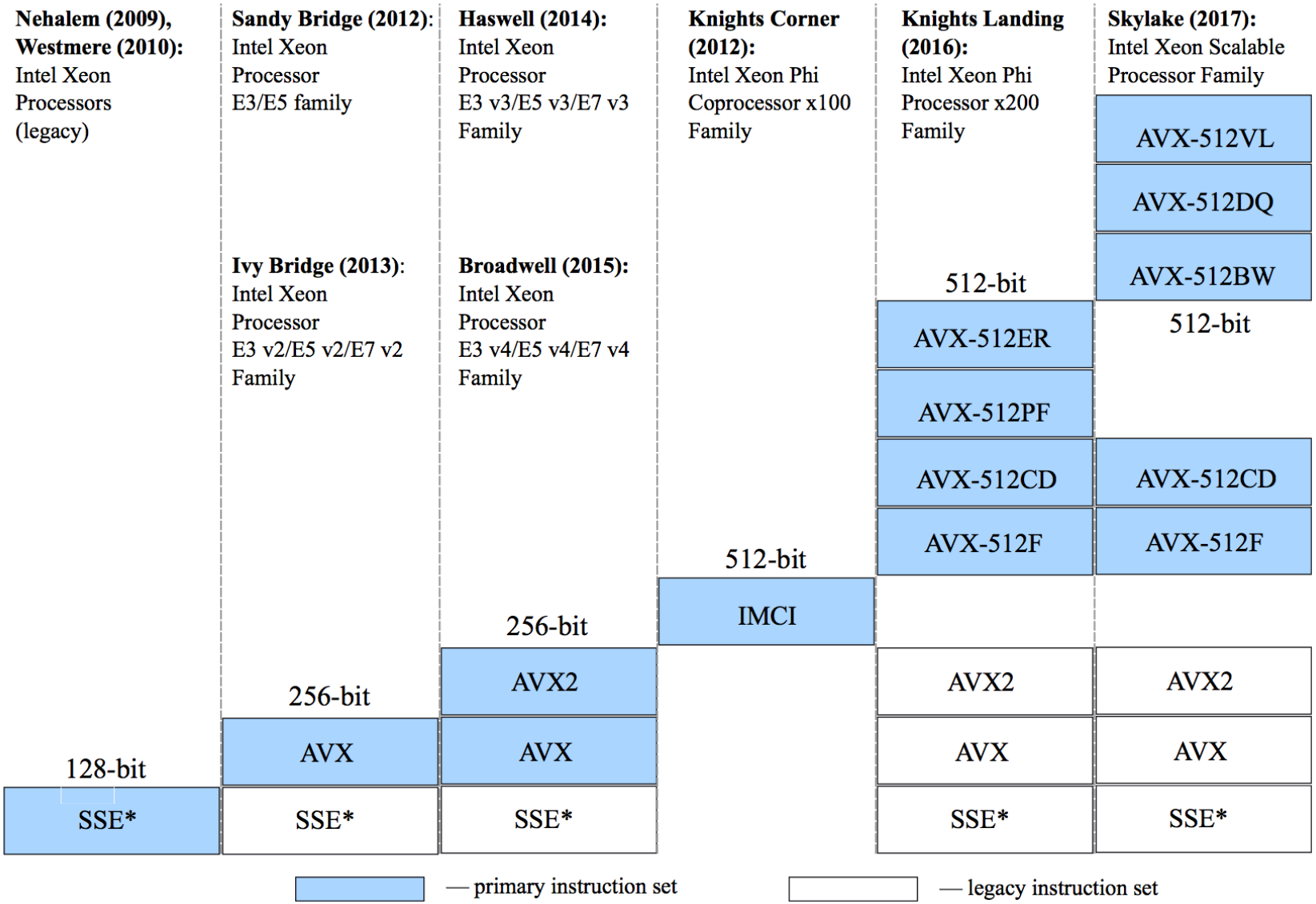
Function Call Sites and Loops	CPU Time		Type
	Total Time	Se...	
[loop in scamp._omp_fn.0 at scamp_vectorized.cpp:100]	13,668s 99,3%	7,6 ...	Scalar
[loop in scamp._omp_fn.0 at scamp_vectorized.cpp:116]	3,215s	3,2 ...	Scalar
[loop in scamp._omp_fn.0 at scamp_vectorized.cpp:110]	2,799s	2,7 ...	Scalar
[loop in scamp._omp_fn.0 at scamp_vectorized.cpp:88]	0,030s	0,0 ...	Vectorized (B...
[loop in scamp._omp_fn.0 at scamp_vectorized.cpp:88]	13,710s 99,6%	0,0 ...	Scalar
[loop in main at scamp_vectorized.cpp:280]	0,010s	0,0 ...	Scalar

Observación

Encontramos que la autovectorización (e.g., con OpenMP) **no es capaz de explotar todo el potencial SIMD de los algoritmos**

Oportunidad

- **Vectorización explícita** usando AVX2 y AVX-512



Propuesta

- Este trabajo se centra en optimizar los algoritmos Matrix Profile utilizando los **intrínsecos AVX** disponibles para CPUs Intel
  - Native instructions (one-to-one with assembly)
    - `_mm_load_ps()` ↔ `movaps`
    - `_mm_add_ps()` ↔ `addps`
    - `_mm_mul_pd()` ↔ `mulpd`
    - `_mm256_load_pd()` ↔ `vmovapd`
    - `_mm256_add_pd()` ↔ `vaddpd`
    - `_mm256_mul_pd()` ↔ `vmulpd`
    - …
  - Multi instructions (map to several assembly instructions)
    - `_mm_set_ps()`, `_mm_set1_ps()`, …
    - `_mm256_set_pd()`, `_mm256_set1_pd()`, …
  - Macros and helpers
    - `_MM_TRANSPOSE4_PS()`
    - `_MM_SHUFFLE()`
    - …

Contenido

Matrix Profile Autovectorizado

- Desenrollando el bucle interno de SCRIMP se puede facilitar la autovectorización a OpenMP **a lo largo de la diagonal**
- Sin embargo, se requiere de un **paso secuencial** para satisfacer dependencias

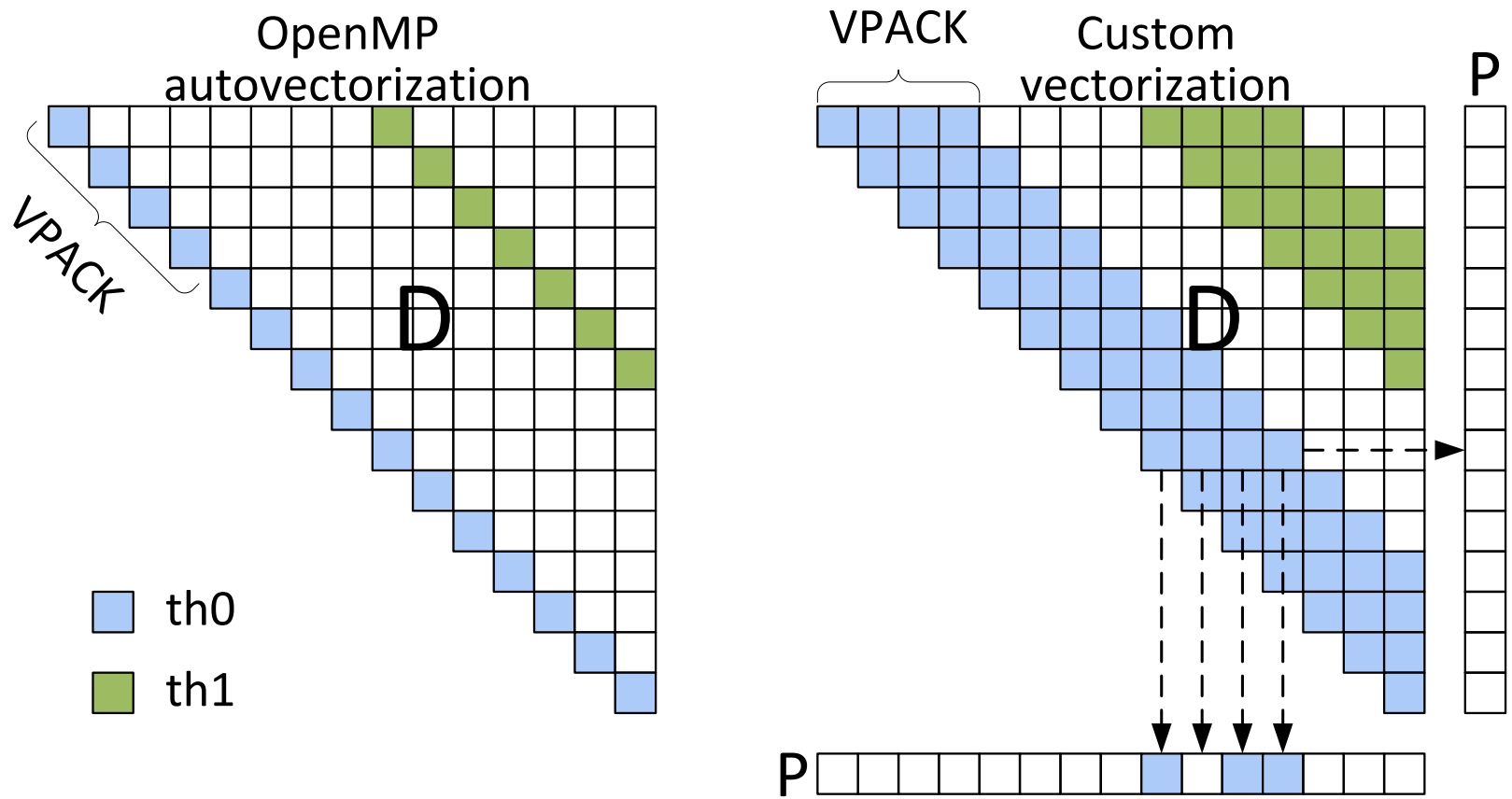
---

1: **for** $k \leftarrow 0$ to $size - 1$ **do** ▷ *Bucle original*
2: $\quad qs_k \leftarrow qs_{k-1} + t_{i+m-1+k}t_{j+m-1+k} - t_{i-1+k}t_{j-1+k}$;
3: $\quad ds_k \leftarrow dist(m, qs_k, \mu_{i+k}, \sigma_{i+k}, \mu_{j+k}, \sigma_{j+k})$;

1: **for** $k \leftarrow 0$ to $vectFact - 1$ **do** ▷ *Vectorizado*
2: $\quad qs_k \leftarrow t_{i+m-1+k}t_{j+m-1+k} - t_{i-1+k}t_{j-1+k}$;
3: $qs_0 \leftarrow qs_0 + q$
4: **for** $k \leftarrow 1$ to $vectFact - 1$ **do** ▷ *Secuencial (dependencia)*
5: $\quad qs_k \leftarrow qs_k + qs_{k-1}$;
6: $q \leftarrow qs_{vectFact-1}$;
7: **for** $k \leftarrow 0$ to $vectFact - 1$ **do** ▷ *Vectorizado*
8: $\quad ds_k \leftarrow dist(m, qs_k, \mu_{i+k}, \sigma_{i+k}, \mu_{j+k}, \sigma_{j+k})$;

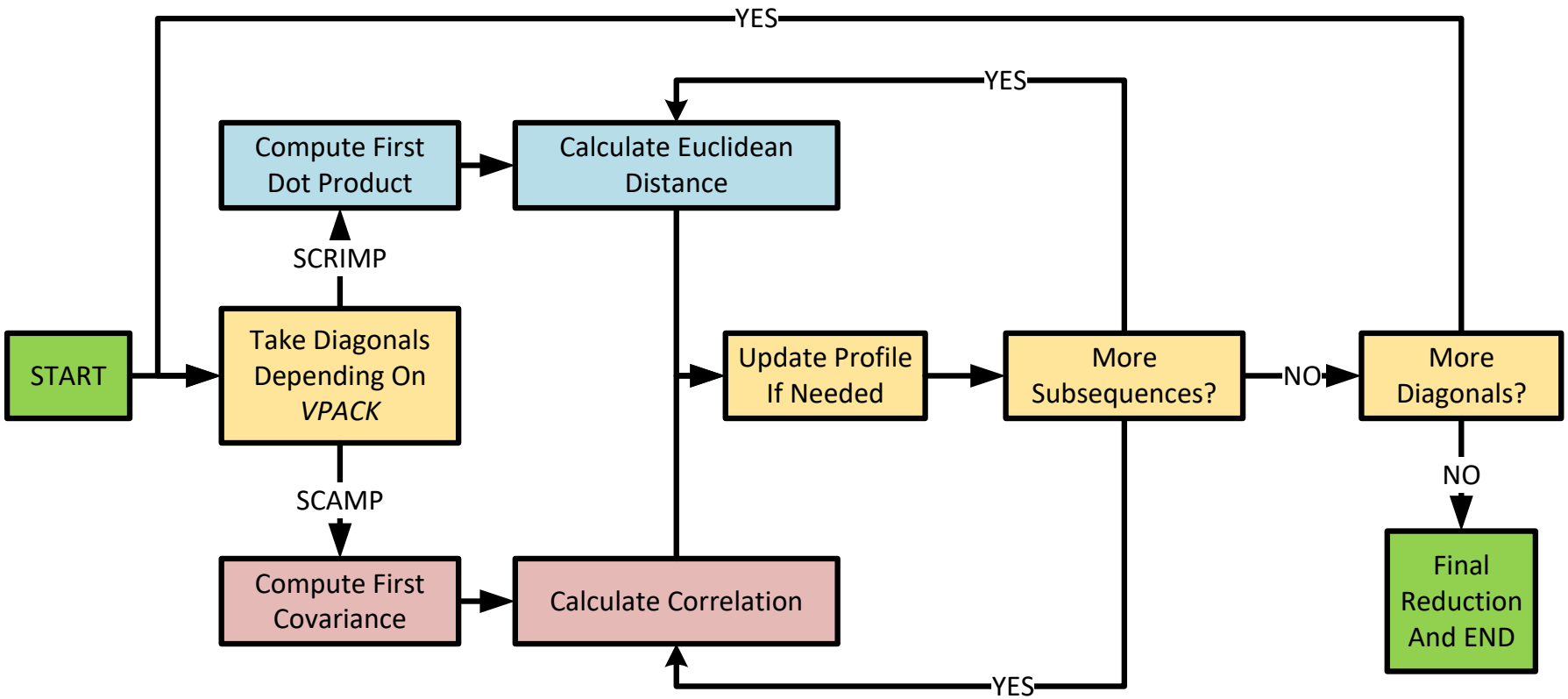
Matrix Profile con Vectorización Explícita

- En lugar de vectorizar la diagonal, **cada hilo computa varias diagonales a la vez** como se muestra en la figura

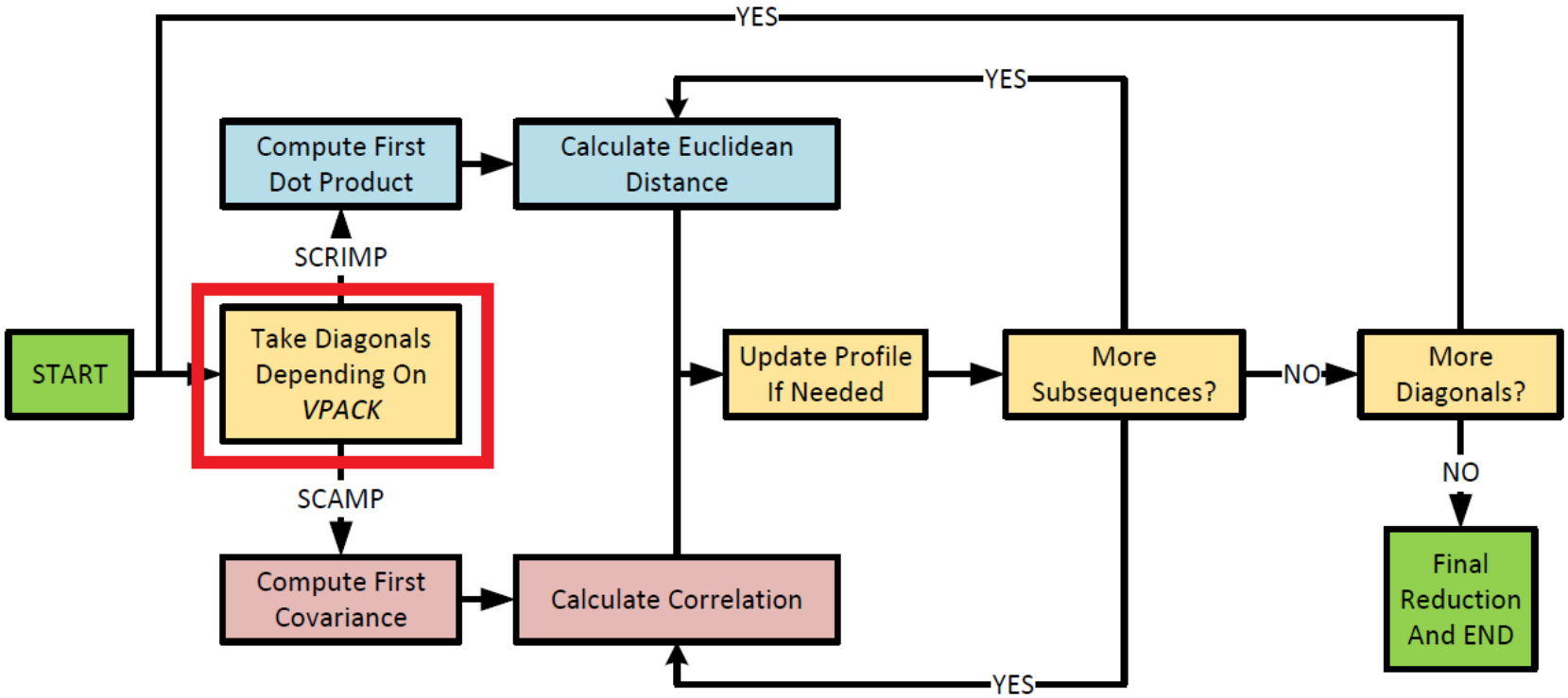


Implementación

- Algoritmo



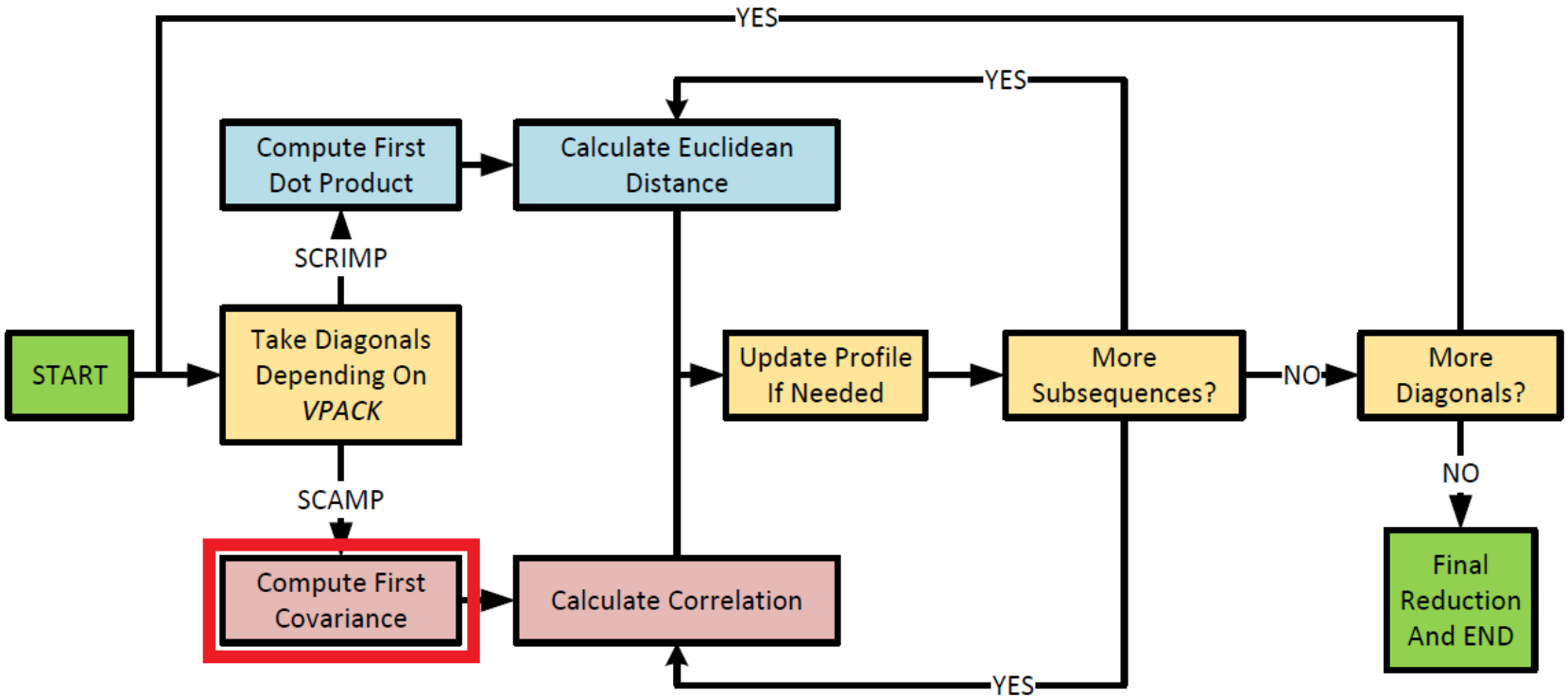
Implementación



- 1) Tomar VPACK diagonales

```
#pragma omp for schedule(dynamic)
for (ITYPE diag = exclusionZone + 1; diag < profileLength; diag +=
↪  VPACK)
```

Implementación

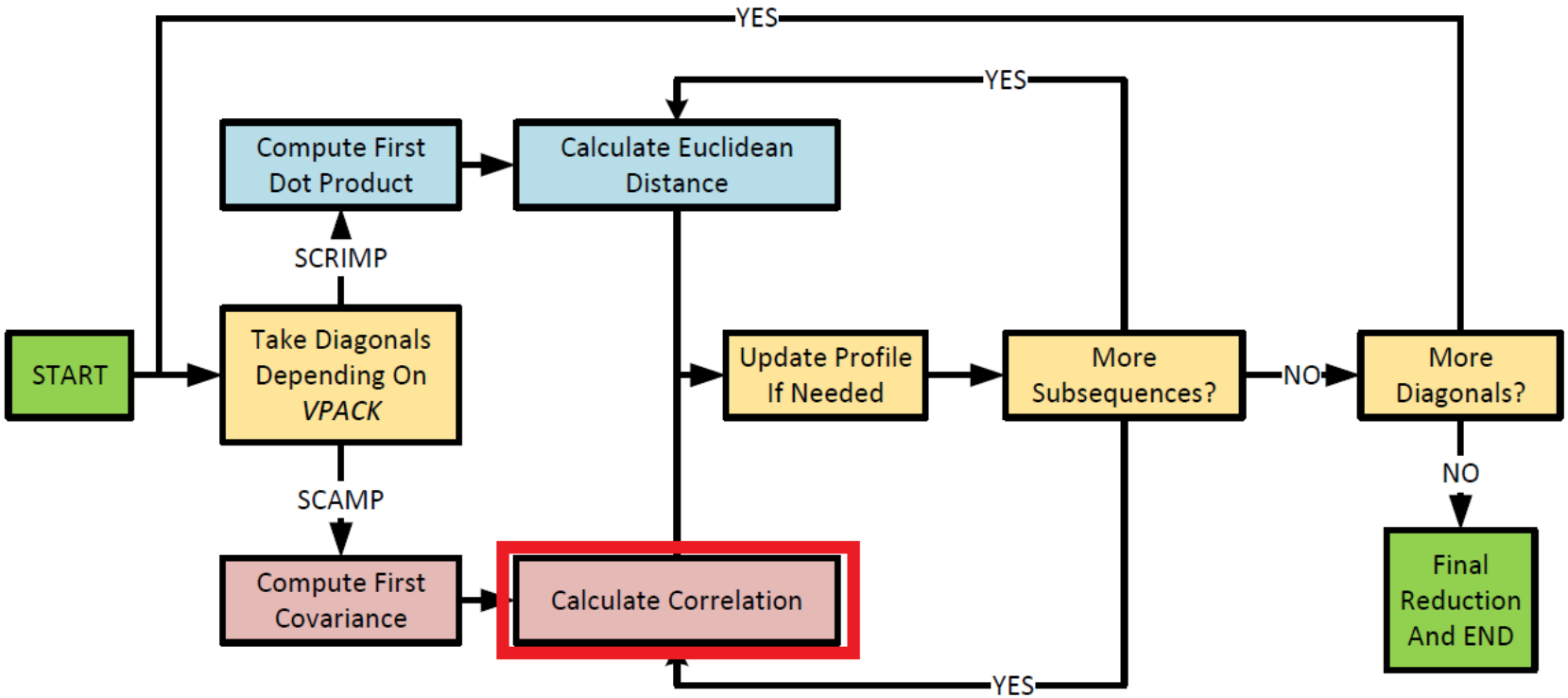


Implementación

- 2) Computar las primeras covarianzas

```
VDTYPE covariance_v = SETZERO_PD();
for (ITYPE i = 0; i < windowSize; i++) {
  VDTYPE tSeriesWinDiag_v = LOADU_PD(&tSeries[diag + i]);
  VDTYPE meansWinDiag_v = LOADU_PD(&means[diag]);
  VDTYPE tSeriesWin0_v = SET1_PD(tSeries[i]);
  VDTYPE meansWin0_v = SET1_PD(means[0]);
  covariance_v = FMADD_PD(SUB_PD(tSeriesWinDiag_v, meansWinDiag_v),
  ↪  SUB_PD(tSeriesWin0_v, meansWin0_v), covariance_v);
}
```

Implementación

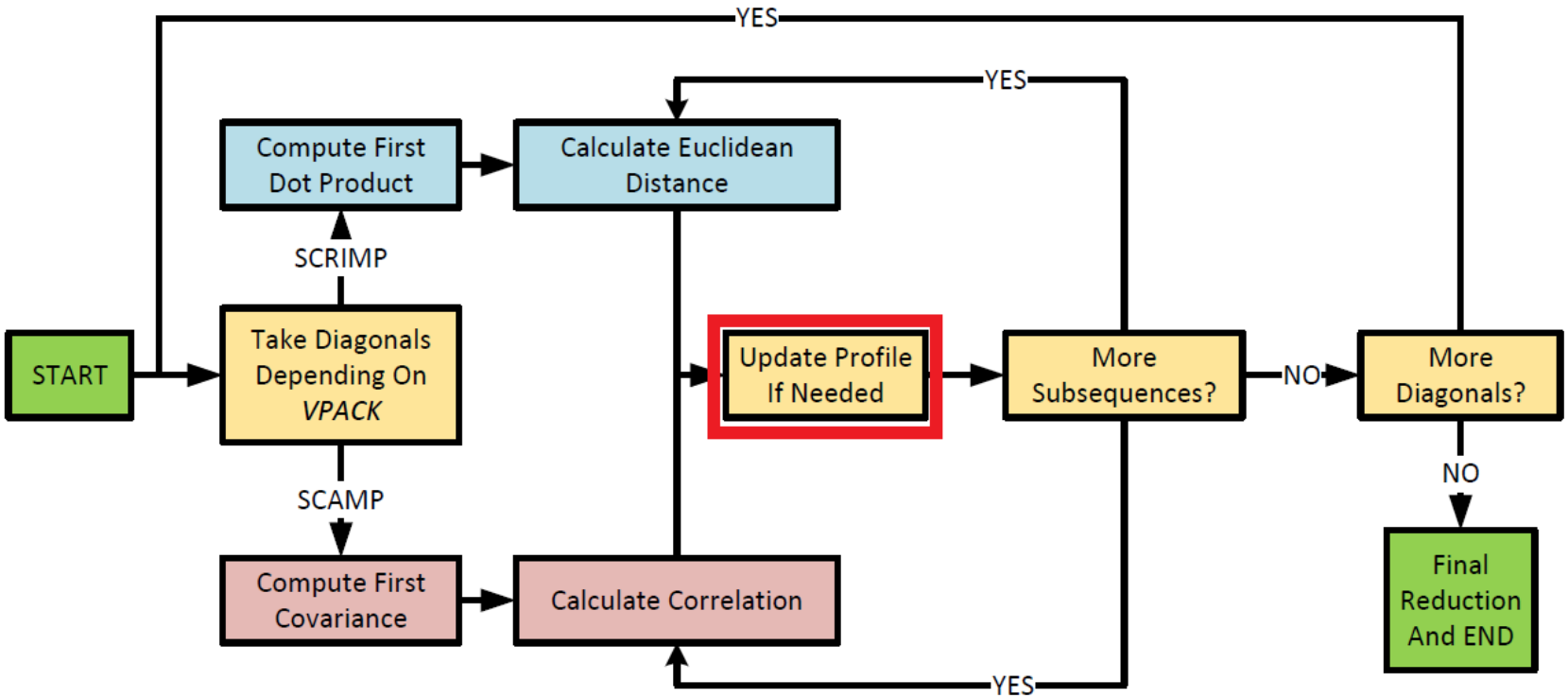


Implementación

- 3) Computar las correlaciones

```
normsi_v = _mm512_set1_pd(norms[i]);
normsj_v = _mm512_loadu_pd(&norms[j]);
correlation_v = _mm512_mul_pd(_mm512_mul_pd(covariance_v,
↪  normsi_v), normsj_v);
```

Implementación

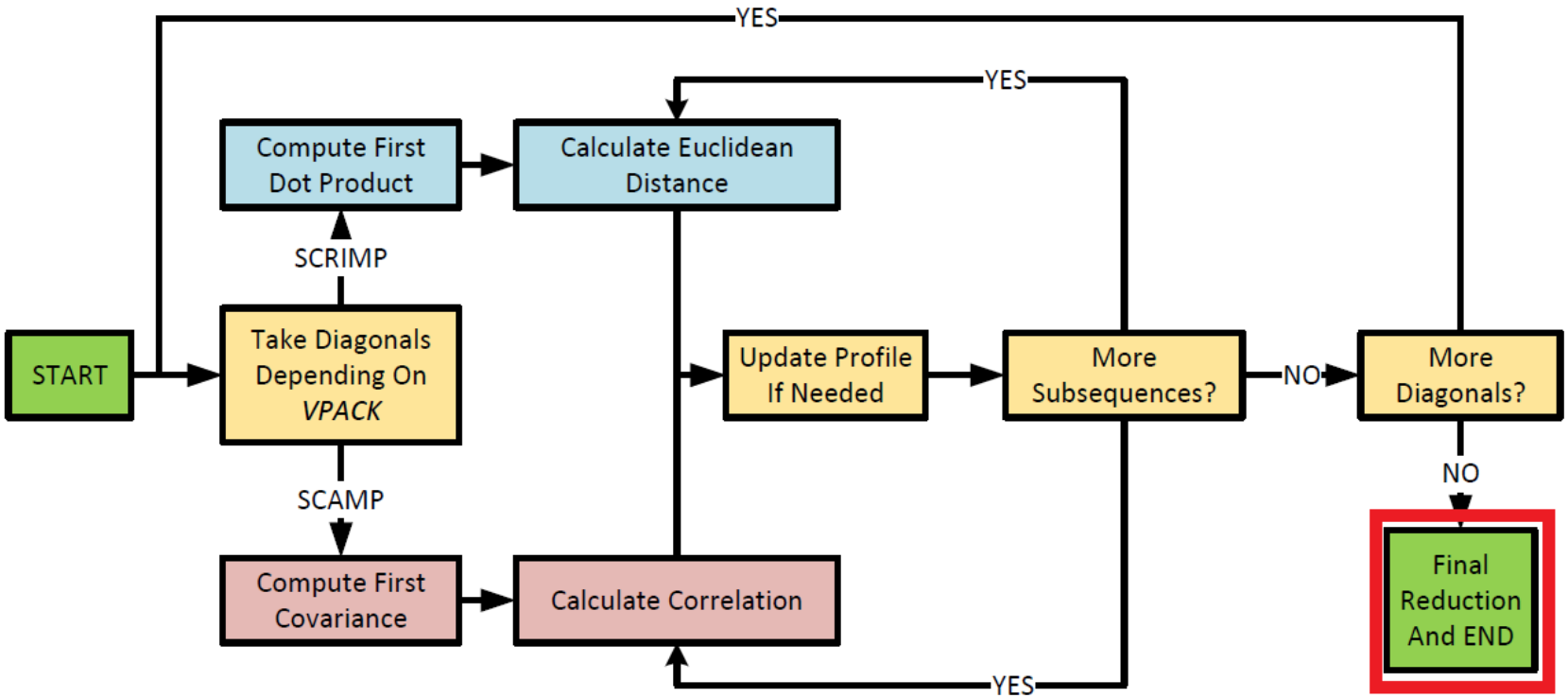


Implementación (II)

- 4) Actualizar el Profile si es necesario

```
for (int ii = 0; ii < VPACK; ii++) {
  if (correlation[ii] > profile_tmp[i + my_offset]) {
    profile_tmp[i + my_offset] = correlation[ii];
    profileIndex_tmp[i + my_offset] = j + ii;
    }
}
VDTYPE profilej_v = LOADU_PD(&profile_tmp[j + my_offset]);
VMTYPE mask = CMP_PD(correlation_v, profilej_v, _CMP_GT_OQ);
MASKSTOREU_PD(&profile_tmp[j + my_offset], mask, correlation_v);
MASKSTOREU_EPI(&profileIndex_tmp[j + my_offset], mask, SET1_EPI(i));
```

Implementación



Implementación

- 5) Reducción final

```
#pragma omp for schedule(static)
for (ITYPE colum = 0; colum < profileLength; colum += VPACK) {
  VDTYPE max_corr_v = SET1_PD(-numeric_limits<DTYPE>::infinity());
  VITYPE max_indices_v;
  for (ITYPE th = 0; th < numThreads; th++) {
    VDTYPE profile_tmp_v = LOADU_PD(&profile_tmp[colum + (th *
    ↪  (profileLength + VPACK))]);
    VITYPE profileIndex_tmp_v = LOADU_SI((VITYPE
    ↪  *)&profileIndex_tmp[colum + (th * (profileLength +
    ↪  VPACK))]);
    VMTYPE mask = CMP_PD(profile_tmp_v, max_corr_v, _CMP_GT_OQ);
    max_indices_v = BLEND_EPI(max_indices_v, profileIndex_tmp_v,
    ↪  mask);
    max_corr_v = BLEND_PD(max_corr_v, profile_tmp_v, mask);
  }
  STORE_PD(&profile[colum], max_corr_v);
  STORE_SI(&profileIndex[colum], max_indices_v);
}
```

Contenido

Metodología

- Usamos un servidor con cuatro CPUs **Intel Xeon Gold** 5218 (4x16 cores, 4x32 threads)

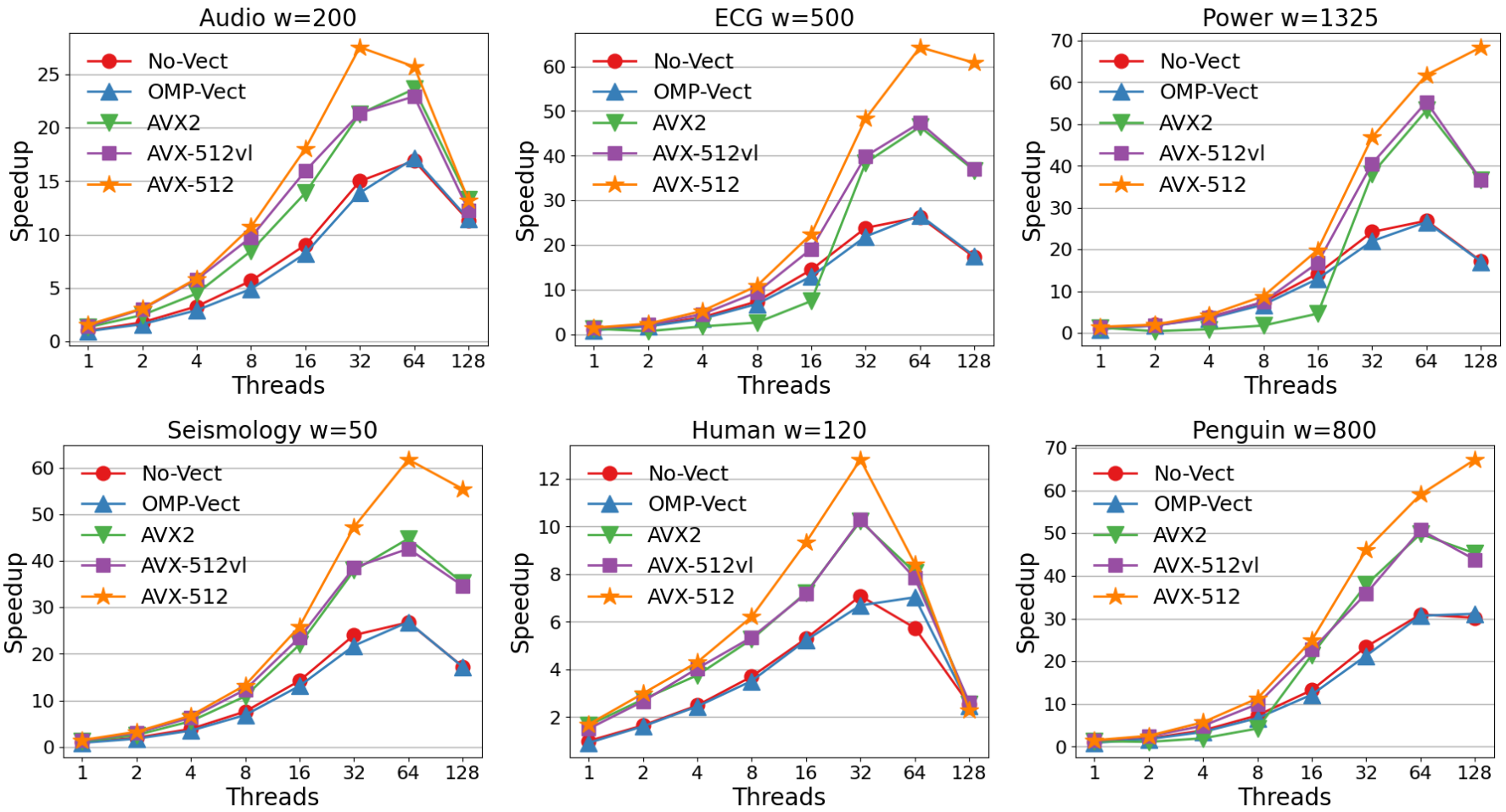
- Las series temporales analizadas son:

Time Series	**n**	**m**	**Max**	**Min**
Audio	20.234	200 (2s)	6,69	-56,48
ECG	180.000	500 (2s)	2,6	0,325
Power	180.000	1.325 (8h)	140	0
Seismology	180.000	50 (2,5s)	696,1	-185.7
Human Act.	6.997	120 (12s)	2,51	-2,9
Peng. Behav.	109.842	800	0,52	-0,21

- Series largas:

Time Series	**n**	**m**	**Max**	**Min**
ECG	1.800.000	500 (2s)	3,39	-1,635
Power	1.859.587	1.325 (8h)	140	0
Seismology	1.728.000	50 (2,5s)	2329,3	-2334,6

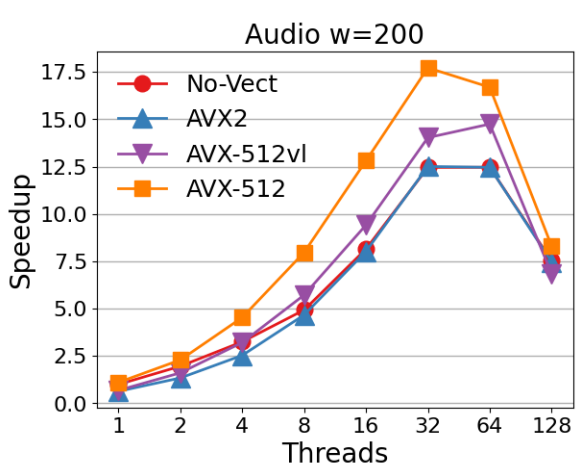
SCRIMP

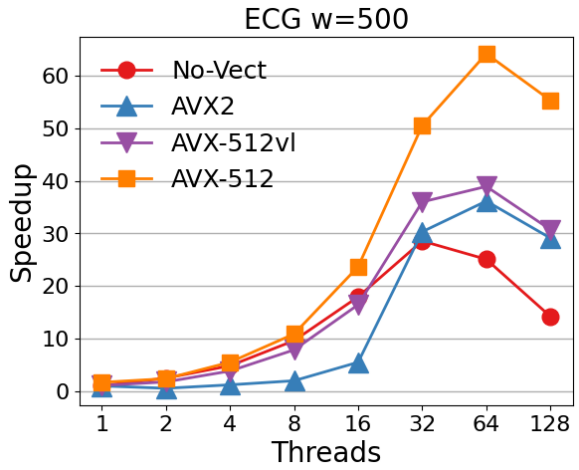


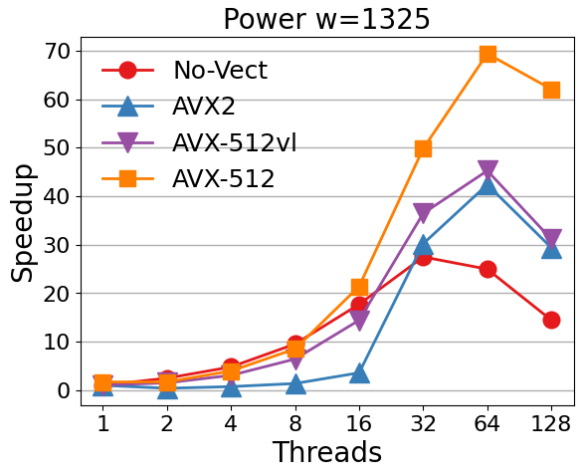
Observación

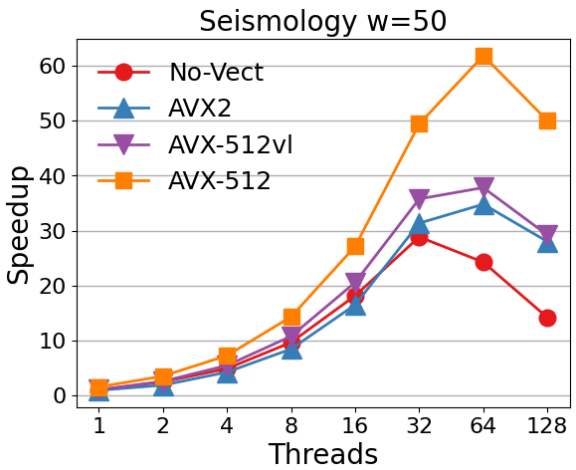
La autovectorización de `OMPVect` es peor que AVX2 excepto para series con ventanas grandes y menos de 16 threads.

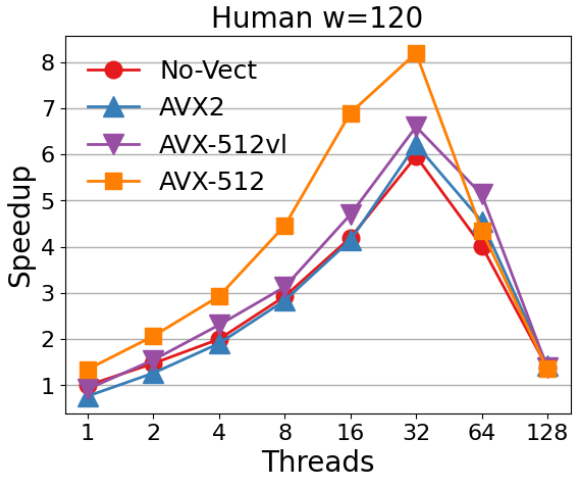
SCAMP

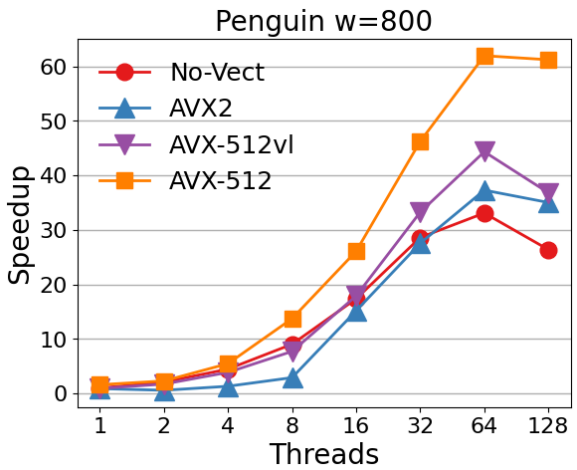












Observación

Las series pequeñas (Audio y Human) muestran mala escalabilidad debido al desbalanceo de carga con un nº de threads alto

Problema con las máscaras

- La instrucción `maskstore` se traduce a `vmaskmov` y `vmovup {mask}` en AVX2 y AVX-512, respectivamente

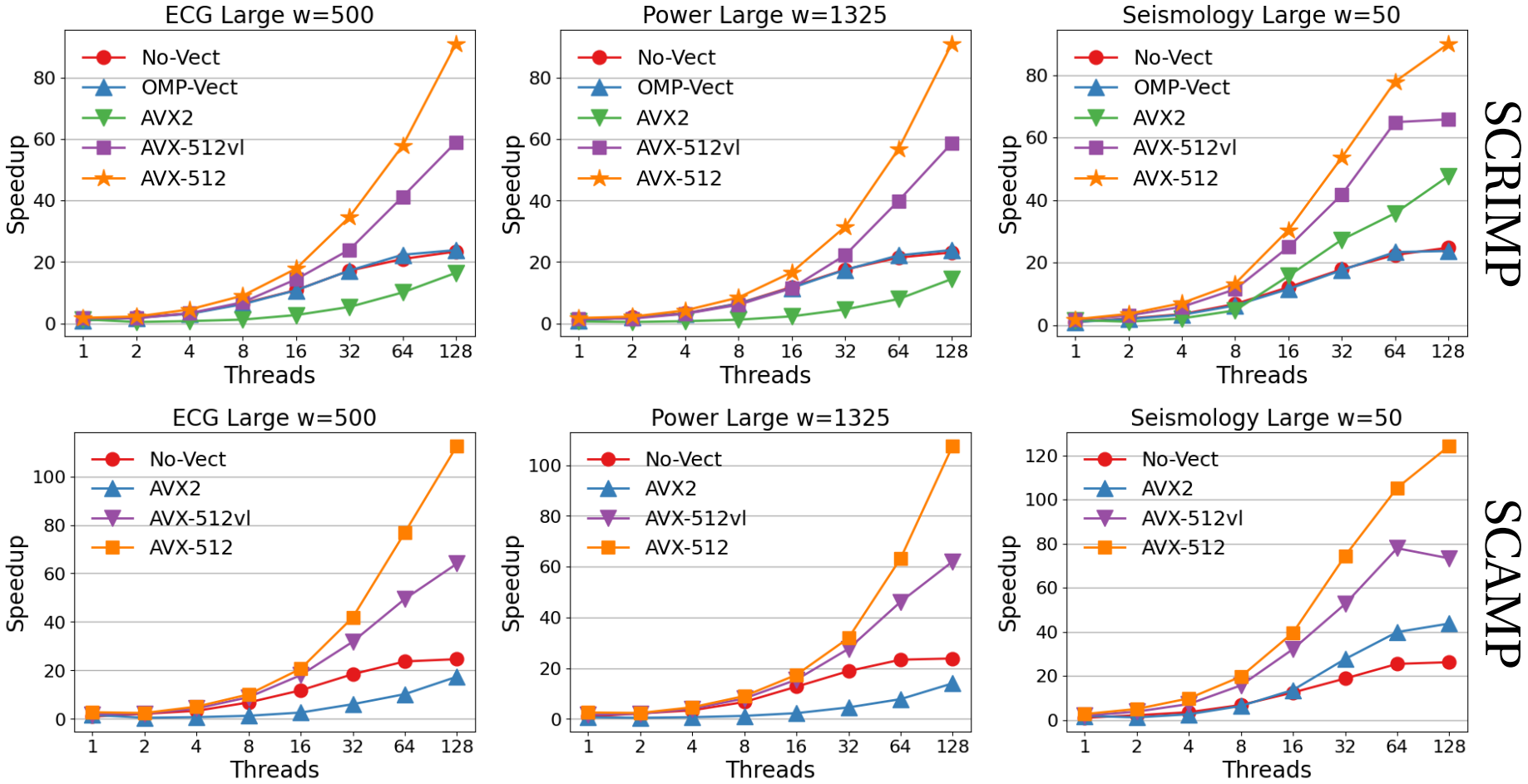
				# dTLB store misses		
Impl.	**Series**	**n**	**m**	**#th=2**	**#th=16**	**#th=32**
AVX2	Power	180K	1.325	9.320.502.463	5.209.871.973	24.371.203
AVX-512vl				3.903.164.142	1.949.028.376	9.241.465
AVX-512				1.874.597.618	744.614.596	1.868.692
AVX2	Seismo	180K	50	144.912.286	60.832.193	13.134.275
AVX-512vl				57.375.614	44.190.128	6.439.342
AVX-512				40.623.017	31.620.232	2.220.071

- Manual de Intel: En casos en los que los bits de máscara indiquen que los datos no deben ser actualizados, los bits A y D de la página se actualizarán dependiendo de la implementación

Observación

Para un número de threads menor o igual que 16 sólo se utiliza una CPU y la TLB muestra muchos fallos que lanzan *assists*

Series largas



Observación

Las series largas muestran buena escalabilidad por balanceo de carga y el problema con las máscaras se mantiene a partir de 16 threads

Contenido

Conclusions and Future Work

- Estudiamos los beneficios de usar **vectorización explicita** para el análisis de series temporales
- Se proponen las implementaciones `SCRIMPvect` y `SCAMPvect` que hacen uso de extensiones vectoriales y se comparan con soluciones de **autovectorization**
- La evaluación experimental muestra mejoras de hasta $4\times$ con respecto a la autovectorization
- Como trabajo futuro se propone el estudio de la vectorización de algoritmos de Matrix Profile con **Dynamic Time Warping**

Exploiting Vector Extensions to Accelerate Time Series Analysis

Ricardo Quislant Ivan Fernandez Eduardo Serralvo
Eladio Gutierrez Oscar Plata

Department of Computer Architecture
University of Malaga (Spain)
quislant@uma.es

XXXIII Jornadas de Paralelismo